Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
im Fachbereich Mathematik und Informatik
der Freien Universität Berlin

ScatterClipse: Architekturzentrierte Plattform und Werkzeuge für Entwicklung, Test und Verwaltung drahtloser Sensornetze

---

Disputationsvortrag: 12.02.2010

Bearbeitet von:
Mohammad Al Saad
Kniephofstraße 23a
12157 Berlin

Gutachter:
Prof. Dr.-Ing. Jochen Schiller
Prof. Dr. rer. nat. habil. Jochen Seitz

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 9. November 2009

Mohammad Al Saad

Danksagung

Zuerst danke ich Herrn Prof. Schiller, dass er es ermöglicht hat, dieses Projekt zu verfolgen, und mich während meiner Promotion betreut und begleitet hat. Des Weiteren danke ich mich bei Frau Prof. Fehr für ihre Unterstützung und Zusammenarbeit in den letzten fünf Jahren, in denen ich diese Arbeit abschließen konnte. Mein Dank geht auch an Herrn Prof. Seitz für seine Bereitschaft das Zweitgutachten für die Dissertation zu erstellen. Dr. Hartmut Ritter danke ich seine Mitwirkung an der Entstehung dieser Arbeit.

Mit Liebe danke ich meiner Mutter, die mir aus der Ferne beigestanden hat und in Gedanken immer bei mir war, wie auch meine Geschwister.

Ebenso geht mein Dank an meine Schwiegereltern, die mich moralisch während meiner Promotion unterstützt haben.

Zuletzt danke ich meiner Frau Silvie, die mich während meiner gesamten Promotion unterstützt hat und mir mit Liebe und viel Geduld zur Seite stand. Bei ihr kann ich mich nicht genug bedanken. Ein Dicker Kuss geht an meinen Sohn Khaloude, aus dessen Lächeln ich Freude schöpfe.

Es bleibt mein Vater, der uns vor 8 Jahren verlassen hat. Ihm widme ich diese Arbeit in Liebe und Dankbarkeit.

Zusammenfassung

ScatterClipse ist eine plugin-orientierte Werkzeugkette für automatisiertes Testen, Entwicklung und Verwaltung von auf drahtlosen Sensornetzen laufenden Anwendungen. Das Hauptziel ist die automatisierte Generierung von Software-Systemfamilien für die ScatterWeb Sensorknoten. Für die Automatisierung wird die architekturzentrierte Methode des modellgetriebenen Paradigmas eingesetzt.

Der neue Ansatz von ScatterClipse liegt in der Integration von visueller, automatisierter Fehlersuche und WSN-Verwaltungseigenschaften innerhalb des modellgetriebenen Anwendungsentwicklungsprozesses der Sensorknoten. Zusätzlich zur Automatisierung liegt der Fokus auf dem Aspekt der Mensch-Maschine-Schnittstelle (*human-computer interfaces*) für WSN. So bietet ScatterClipse die Möglichkeit, verschiedene Charakteristika und Merkmale des WSN visuell und interaktiv zu testen und zu steuern. Um die Produktivität der Plattform zu erhöhen, kann auf die Hauptdienste sowohl im lokalen als auch im entfernten internetbasierten Modus zugegriffen werden.

Die Eclipse-basierte, Plugin-orientierte Architektur fördert die Erweiterung der Werkzeugkette, wenn neue Anforderungen an WSN auftauchen. Einerseits kann der Benutzer bestimme Plugins (Funktionalitäten) unabhängig voneinander verwenden, so dass eine Trennung der Belange (*separation of concerns*) erreicht wird. Andererseits kann er gleichzeitig zwischen den verschiedenen Plugins kooperativ navigieren, wodurch Kohärenz erreicht wird. ScatterClipse basiert vollständig auf Open Source-Komponenten, die ohne zusätzliche Kosten erhältlich sind.

Abstract

ScatterClipse is a plugin-oriented toolchain for the automated testing, development and management of applications running on Wireless Sensor Networks. The main goal is the furthermost automated generation of software system families for the ScatterWeb sensor boards. The architecture centric method of the model driven paradigm is used for automation.

The ScatterClipse's novel approach lies in the accomplishment of the integration of visual automated application debugging and WSN-management features in the model-driven application development process of the sensor boards. In addition to automation the focus lies here with the aspect of *human-computer interfaces* for WSN. Thus ScatterClipse provides the opportunity to test and manage different characteristics and properties of WSN visually and interactively. In order to improve the platforms productivity, its main features can be accessed in local as well as in remote, or internet based, mode.

The eclipse-based plug-in oriented architecture encourages the enhancement of the toolchain, when new questions concerning WSN arise. On the one hand the user can operate certain plug-ins (functionalities) independent from each other, so that a "separation of concerns" is achieved, and on the other hand the user can navigate the different plug-ins collaboratively at the same time, whereby coherence is achieved. ScatterClipse is based wholly on open source components, which are available at no extra cost.

Inhaltsverzeichnis

1. Einleitung 1
1.1. Forschungsdiskussion und Anforderungen 2
1.2. Zielsetzung und Vorhaben 4
1.3. Aufbau der Arbeit 5

I *Allgemeines*

2. Verwendete Technologien und Werkzeuge 6
2.1. Das architekturzentrierte modellgetriebene Paradigma 6
2.2. Die Eclipse Plattform 9
2.3. Das Eclipse Modeling (EMF) 11
2.4. Graphical Modeling (GMF) 14
2.5. openArchitectureWare (oAW) 16
2.6. Zusammenfassung 24

II *Die generative Infrastruktur*

3. Der Pfad des Anwendungstests: Visual ScatterUnit 26
3.1. Architektur des Rahmenwerks Visual ScatterUnit 28
3.2. Die Testautomatisierungsschicht ScatterUnit 34
3.3. Die Implementierung des Testfalls in ScatterUnit 36
3.4. Modellgetriebenes Testen in Visual ScatterUnit 39
3.5. Der visuelle automatisierte Debuggingprozess 43
3.6. Zusammenfassung 49
4. Technische Realisierung des Testrahmenwerks 50
4.1. Entwurf und Realisierung der Domänensprache (DSL) 50
4.2. Live-Validierung der Modelle 58
4.3. Die Codegenerierung 62
4.4. Die Dienste der Testschicht ScatterUnit 65
4.5. Zusammenfassung 69
5. ScatterFactory: Der Pfad zur Anwendungsentwicklung 71
5.1. Das Metamodell 72
5.2. Die Modellierungsebene 74
5.3. Die Modellvalidierung 77
5.4. Die Codegenerierung 78
5.5. Kompilierung und Initialisierung der Software (Deployment) 82
5.6. ScatterFactory2 82
5.7. Zusammenfassung 86

III *WSN-Verwaltung*

6. ScatterPlug: Der Pfad zur WSN-Visualisierung und -Analyse 88
6.1. Die Kommunikationsschicht 88
6.2. Das Root Plugin 92
6.3. Main View 94
6.4. Das Statistik View 95
6.5. Zusammenfassung 98
7. ScatterEditor: Der Pfad zur WSN Konfiguration, Steuerung und Überwachung 99
7.1. Architektur und Design von ScatterEditor 99
7.2. Dienste und Komponenten von ScatterEditor 101
7.2.1. Die Verbindungskomponente “Connection” 101
7.2.2. Visualisierung durch den Property-Reiter 103
7.2.3. Das Terminal 104
7.2.4. Aktualisierung der Softwaremodule via Funkkanal „Over The Air Software Updates“ 106
7.2.5. Die WSN-Überwachung (Alarm) 107
7.3. Internetintergration in ScatterEditor 109
7.4. Zusammenfassung 110

IV *Bewertung und Positionierung von ScatterClipse*

8. Beurteilung und Auswertung der Werkzeugkette 112
8.1. Die Codegröße der in ScatterFactory erstellten Anwendungen 112
8.2. Bewertung der Testautomatisierungsschicht 114
8.3. Bewertung des modellgetriebenen Anwendungstests (Fallstudie) 119
8.3.1. Analyse und Evaluierung der Fallstudie 129
8.3.2. Schlussbemerkung zur Fallstudie 135
9. Verwandte Arbeiten 134
9.1. Zusammenfassung 145
10. Fazit und Zusammenführung 147

11. Anhang 152
11.1. Abkürzungsverzeichnis 152
11.2. Glossar 153
11.3. Benutzerinteraktion 155
11.3.1. Der Wizard zur Einrichtung des WSN 155
11.3.2. Das ScatterFlash-Plugin: Kompilierung und Initialisierung der Software 156
11.3.3. OTA-Flashing 157
11.3.4. Der Alarm-Reiter 158

11.3.5. Benutzerinteraktion im entfernten Modus 159
11.4. Das Metamodell der Domänensprache 160
11.5. Die in Checks formulierten Randbedingungen 162
11.5.1. Randbedingungen in Visual ScatterUnit 162
11.5.2. Randbedingungen in ScatterFactory2 167
11.6. ScatterClipse im Einsatz 170
11.7. Verzeichnisse 171
11.7.1. Abbildungsverzeichnis 171
11.7.2. Programmauszugverzeichnis 174
11.7.3. Tabellenverzeichnis 175
11.8. Literaturverzeichnis 176
11.9. Aus der Arbeit entstandene Publikationen 184

1. Einleitung

Drahtlose Sensornetze (Wireless Sensor Networks: WSN) sind in den letzten Jahren Gegenstand erheblicher Forschungstätigkeit gewesen. Sie bestehen aus einer großen Anzahl von Sensoren, die mit Mikro-Kontroller, Datenspeicher, Sensorik und Funkmodulen ausgestattet sind. Ihre Energieversorgung erfolgt durch eine Batterie oder durch Mechanismen, mit denen Energie aus der Umwelt bezogen wird [Schi05]. Die meisten WSN-Anwendungen haben die Überwachung oder Erkennung bestimmter Phänomene zum Ziel. So werden in einer Vielzahl von Anwendungen zur Überwachung des Lebensraums von Wildtieren [Naum08], Haushaltsautomatisierung, Waldbranderkennung [Garc08] und zur militärischen Überwachung WSNs eingesetzt. Die rückgängigen Kosten für Sensorknoten ermöglichen den Einsatz von WSN-Anwendungen, die größere Räume mit einer höheren Knotendichte abdecken. Der Einsatz von Sensornetzwerken erfolgt bereits mit großangelegten Netzwerken, die aus Hunderten oder sogar Tausenden von Knoten bestehen. Das Spektrum für den Einsatz drahtloser Sensornetze wird im Laufe der Zeit noch komplexer und vielfältiger [Kuor05]. Diese Steigerung von Maßstab und Komplexität machen die Anwendungsentwicklung für WSN zeitaufwändiger und fehleranfälliger. Die Herausforderungen bezüglich Programmierung und Testen von WSN-Anwendungen und damit die Ansprüche an die dazugehörigen Entwicklungsplattformen steigen entsprechend.

Je anspruchsvoller die Verwendung der ressourcenschwachen drahtlosen Sensornetze wird, desto robuster, zuverlässiger und effizienter muss die Entwicklung ihrer Softwareanwendungen vonstatten gehen. Unerlässlich wird die Optimierung der Anwendungsentwicklung vor dem Hintergrund der erwähnten Steigerung der Anzahl von Knoten, die ein solches Netz bilden. Dies erhöht die Komplexität der Randbedingungen, unter denen die Sensorknoten betrieben werden, was dazu führt, dass die technischen Belange zur Entwicklung der auf den Sensorknoten laufenden Software entsprechend vielschichtiger werden. Ferner bedarf eine robuste Anwendungsentwicklung einer adäquaten Betriebsnähe hinsichtlich der vielfältigen von den Sensorknoten zu bewältigenden Aufgaben.

Gegenwärtige Techniken zur Implementierung dieser Art von Systemen beruhen meist auf simulativen Einsatzszenarien. Daher ist die Entwicklung solcher Simulationswerkzeuge ein Bereich, in dem entsprechend viel geforscht wird. Dabei steht jedoch zumeist das Verhalten des Sensornetzes selbst im Mittelpunkt und weniger jene Entität, welche dieses Verhalten bewirkt, nämlich die auf den Sensorknoten des Funknetzes laufende Software. Statt sich auf Simulationsumgebungen zu beschränken, muss die Anwendungsentwicklung realitätsnäher vonstatten gehen. Geht man noch einen Schritt zurück, so muss zunächst gewährleistet werden, dass die Sensorsoftware bereits bei ihrer Implementierung leistungsfähig und robust erstellt und getestet werden kann und zwar auf den sich bereits im Betrieb befindenden Sensorknoten. Hier mangelt es an entsprechenden Lösungsansätzen und Werkzeugen, was die Notwendigkeit der vorliegenden Arbeit unterstreicht, deren Ergebnis die modellgetriebene ScatterClipse-Werkzeugkette ist. Die Entwicklungsarbeiten an ScatterClipse begannen bereits im Mai 2005, zu einem Zeitpunkt, als diese Notwendigkeit in der Forschung gerade erst erkannt und diskutiert wurde [vgl. Röme04, Ulbr05a].

1.1. Forschungsdiskussionen und Anforderungen

Auf das Fehlen adäquater Programmierungsansätze und entsprechender Werkzeuge angesichts der steigenden Komplexität des Einsatzes von Sensornetzen machen z. B. Wada et al. in ihrem im November 2007 erschienenen Beitrag [Wada07] aufmerksam: „The complexity of WSN application development derives from two major issues: (1) a lack of adequate abstractions in application development, and (2) a lack of coherent tool chains for application development." [Wada07]. Daher stellen sie ein modellgetriebenes Werkzeug zur Anwendungserstellung drahtloser Sensornetze vor, auf das in Kapitel 9 bei der Behandlung verwandter Arbeiten näher eingegangen wird. Diverse weitere Beiträge der Forschung weisen auf die wachsende Notwendigkeit integrierter Werkzeuge zur leistungsfähigeren realitätsnahen Entwicklung von auf den Sensorknoten laufenden Softwareanwendungen hin: „What type of software is needed (…) and what requirements have to be met?" [Röme04]. „Außerdem sollen weitere Schnittstellen und Entwicklungsunterstützung angeboten werden. Was weniger dringend benötigt wird, sind neue Algorithmen für Routing, die eine Verbesserung von wenigen Prozent versprechen." [Schi07].

Diesen Gegebenheiten bzw. Anforderungen müssen die Entwicklungswerkzeuge und verwendeten Ansätze und Technologien Rechnung tragen. Daher verwundert es nicht, dass Softwarehersteller ebenfalls diese Bedürfnisse erkannt haben und bestrebt sind, die WSN-Technologie und deren Softwareentwicklungsprozess[1] in ihre Entwicklungsumgebungen nahtlos zu integrieren. Dies wird z. B. bei Forschungsarbeiten von Microsoft-Research deutlich: „Up until now, research in the area of WSNs has mostly focused on hardware design, self-organisation, a plethora of routing algorithms, or energy saving patterns. The upcoming challenge is the design of efficient interfaces to the WSNs and their tight integration into development environments such as Visual Studio." [Naum07]

Der steigenden Komplexität sollten Werkzeuge mit entsprechender Abstraktion während der Anwendungsentwicklung entgegenwirken, wodurch die Fachlichkeit in den Mittelpunkt rückt, und von den technischen Belangen abstrahiert werden kann. Dies ist auch der erste Aspekt, der im bereits erwähnten Beitrag von Wada et al. bei den meisten vorhandenen Entwicklungswerkzeugen für WSN-Anwendungen bemängelt wird: „The first issue is a lack of adequate abstractions that application developers can leverage for the rapid implementation of WSN applications. The level of abstraction remains very low in the current practice of WSN application development." [Wada07].

Die Notwendigkeit zur Abstraktion von technischen Belangen wird oft mittels einer gesondert entwickelten Middleware-Schicht bewerkstelligt. Jedoch bleibt auch dann eine effektive Softwaremethodik und die dazugehörigen Werkzeuge zur Anwendungsentwicklung unverzichtbar: "However, middleware trades off resource consumption for abstraction and programming convenience. Some devices might not even be capable of hosting a middleware layer. Therefore, application development for AS-NETs requires an extended software development methodology and tools in addition to existing software engineering techniques." [Ulbr05a]

Eine weitere Anforderung besteht hinsichtlich unterstützender Funktionalitäten, welche die erwarteten Werkzeuge dem Benutzer zur Verfügung stellen sollen, um ihn bei der Bewältigung der wachsenden Komplexität zu unterstützen: „In addition

[1] Siehe Glossar.

to programming, WSN application development involves a series of labor-intensive tasks. (…) Application developers need to manually work on these tasks in ad-hoc manners with various tools. Those tools are often not interoperable and not well chained to improve developers' productivity. As a result, it takes considerable time through a long sequence of tasks until developers run their applications. The agility of WSN application development remains very low." [Wada07] Den Aspekt der Optimierung des Softwareentwicklungsprozesses drahtloser Sensornetze betonen auch Losilla et al. in ihrem im September 2007 veröffentlichten Beitrag [Losi07], in dem sie ihr modellgetriebenes Werkzeug vorstellen, mit dessen Hilfe Sensoranwendungen komponentenbasiert unter Verwendung von UML[2] [WebUML] erstellt werden. „Current proposals for WSN application development are mainly focused on implementation issues. (…) The lack of a Software Engineering methodology which supports the entire development life-cycle of these applications, commonly results in highly platform-dependent designs, difficult to maintain, scale and reuse." [Losi07]

Die Notwendigkeit von Entwicklungswerkzeugen für Anwendungen drahtloser Sensornetze wird auch von Teng et al. in ihrer im Oktober 2008 veröffentlichten Studie über die bis dahin vorhandenen Werkzeuge behandelt.[3] Sie sehen einen hohen Bedarf an leistungsfähigeren Lösungsansätzen zur Programmierung und Verwaltung von WSN-Anwendungen. Die Anforderung an solche Werkzeuge, deren Architektur und die angewandten Paradigmen listen Teng et al. als Schlussfolgerung ihrer Studie auf. So halten sie u.a. fest, dass die Handhabbarkeit des vom Werkzeug verfolgten Mechanismus zur Anwendungsentwicklung der Sensorknoten ein unerlässliches Merkmal sei, über das gegenwärtige Werkzeuge verfügen müssen: "WSN development tools should simplify the development and management through its features." [Teng08] Ferner ist ein hoher Grad an Kohärenz und Zusammenarbeit innerhalb des Werkzeugs zu gewährleisten: "The current state-of-the-art, disjointed approach is the lack of a standard and unified toolset for building WSN applications, and the entire software development process is ad hoc. Thus, WSN development tools should provide an integrated, efficient, and convenient development toolchain for end users." [Teng08] Der Aufbau der Werkzeugkette soll jedoch auch bei einer hohen Integrität eine offene Architektur aufweisen, um zukünftige Erweiterungen nahtlos bewerkstelligen zu können: „WSN development tools should provide a powerful mechanism allowing nearly unlimited extensions and enhancements of its inbuilt functionality for end users." [Teng08]

Diverse Forschungsbeiträge bestätigen demnach die Notwendigkeit von innovativeren, leistungsfähigeren Ansätzen zur Softwareentwicklung für sensitive Sensorknotenanwendungen. Betrachtet man jedoch die bisherigen Beiträge der Forschung, so stellt man fest, dass bis zum jetzigen Zeitpunkt kein Ansatz vorhanden ist, der den Softwareentwicklungsprozess drahtloser Sensornetze *in seiner Gesamtheit betrachtet*. Anwendungsprogrammierung, Verwaltung aber auch Test sollten nicht isoliert von einander, sondern gemeinsam in einem Gefüge behandelt werden. Dies ist das Ziel dieser Arbeit und der Aspekt, in dem sie sich von anderen Ansätzen unterscheidet.

---

[2] Siehe Glossar.

[3] Die Ergebnisse der Studie wurden auf der ICSNC´08 Konferenz vorgestellt, auf der ein großer Teil der Endergebnisse der vorliegenden Arbeit ebenfalls vorgestellt wurden. Daher war es nicht möglich, die ScatterClipse Werkzeugkette in der Studie in Betracht zu ziehen.

1.2. Zielsetzung und Vorhaben

Der Schlüsselansatz hierzu liegt in der Automatisierung des Entwicklungsprozesses, in den gleichzeitig Verwaltungsfunktionalitäten integriert werden, die für den Sensornetzbetrieb notwendig sind. Für die Automatisierung wird der architekturzentrierte Ansatz des modellgetriebenen Paradigmas eingesetzt. Neu ist dabei, dass Modelle nicht nur zum Zweck der Dokumentation bzw. Visualisierung eingesetzt werden. Vielmehr fungieren diese semantikbehafteten und ausdrucksstarken formalen Modelle auch als Technik, um Schlüsselkonzepte und Randbedingungen der Domäne (Plattform) vollständig und prägnant zu repräsentieren. Solche speziellen, aber technologieneutralen Modelle werden in konfigurierbare Codegeneratoren eingefügt, um nach deren Validierung die korrespondierenden Software-Artefakte automatisiert zu generieren und in die entsprechende Plattform (Funksensoren) zu verteilen.

Durch den hohen Grad an Automatisierung beschleunigen sich die Entwicklung und der Test der auf den Sensorknoten laufenden Anwendungen. Desweiteren steigern sich Ersetzbarkeit und Wiederverwendbarkeit der Softwareartefakte, da diese durch die zugehörigen Modellkonstrukte formal repräsentiert werden. Beides erhöht die Produktivität des Entwicklungsprozesses.

Die modellgetriebene Codegenerierung dient weiterhin der maßgeschneiderten Codeerzeugung, durch die nur so viel Code für die Sensorknoten erzeugt wird, wie für ihre jeweilige Rolle erforderlich ist. Dadurch wird nicht nur der knappe Speicherplatz optimiert, sondern es werden unnötige rechen- und energieintensive Softwaremodule vermieden. Der sinkende Anteil an manuellem Code reduziert zugleich die Möglichkeit von Flüchtigkeitsfehlern der Programmierer. Dabei spielt die Validierung auf Modellebene eine wesentliche Rolle, denn je früher die Fehler im Entwicklungsprozess entdeckt werden, desto robuster und verlässlicher wird er.

Zum Zweck der Automatisierung wurde eine generative Infrastruktur (Visual ScatterUnit und ScatterFactory) entwickelt, die das Rückgrat der Plattform bzw. der Werkzeugkette bildet. Zur Modellierung wurde ein graphischer Editor auf Basis des Eclipse Modeling Frameworks [WebEMF] und des Graphical Modeling Frameworks [WebGMF] entwickelt. Für die Überprüfung der Randbedingungen, die an das jeweilige Modell gebunden sind, wurde eine Echtzeit-Validierung in den Editor integriert, um den Entwicklungsprozess robuster zu gestalten. Als Codegenerator wird openArchitecureWare [WeboAW] verwendet, um aus den eingelesenen Modellen den korrespondierenden Code automatisiert zu generieren und diesen dann auf die installierten Sensorknoten auszubringen. Alle verwendeten Rahmenwerke sind Opensource-Projekte der Eclipse-Plattform [WebEcli].

Wie erwähnt, liegt ein weiterer Fokus auf der Integration der für die Verwaltung von Funksensornetzen essentiellen Funktionalitäten in den modellgetriebenen automatisierten Entwicklungsprozess. Dieser soll nicht isoliert gestaltet werden, sondern reibungslos mit Eigenschaften wie Konfiguration, Fehlerbehebung, Überwachung, Nutzerinteraktion sowie der Visualisierung des Sensorstatus (ScatterEditor und ScatterPlug) kombiniert werden. Diese Kombinationsmöglichkeiten sind ein wichtiges Kennzeichen der Plattform, das durch eine Plugin-orientierte Architektur gemäß der Eclipse-Plattform realisiert wird. Der Benutzer kann sich einerseits bestimmter Plugins (Funktionalitäten) unabhängig von den anderen bedienen, wodurch ein *separation of concerns* erreicht wird, andererseits kann er zwischen den verschiedenen Plugins kollaborativ navigieren, was die Kohärenz erhöht.

Um die Produktivität der Plattform zu verbessern, sollen ihre Hauptmerkmale sowohl im lokalen Modus als auch im entfernten Modus bzw. internetbasiert in Anspruch genommen werden können. Damit kann beispielsweise die Verwaltung und Konfiguration auf einem Rechner an einem Ort (Entwicklungs- und Testlabor) erledigt werden, während die Sensoren entfernt an einem anderen Ort (Experimentfeld) unter realen Bedingungen platziert sind. Dies wurde durch eine herkömmliche IP-basierte Client/Server Architektur realisiert.

1.3. Aufbau der Arbeit

Die Kapitel dieser Arbeit sind in vier thematischen Teilen gruppiert. Der erste Teil der Arbeit *Allgemeines* stellt in einem Kapitel die verwendeten Technologien und Werkzeuge vor und erläutert die verschiedenen an der Realisierung der Werkzeugkette beteiligten Werkzeuge sowie deren Zusammenarbeit.

Dieses Kapitel leitet über zum zweiten Teil *Die generative Infrastruktur*, der sich aus drei Kapiteln zusammensetzt, welche die generative Infrastruktur bestehend aus Visual ScatterUnit, ScatterFactory und deren Integration behandeln. Das dritte Kapitel befasst sich mit der modellgetriebenen Testumgebung Visual ScatterUnit zur Automatisierung des Tests der auf den Sensorknoten laufenden Anwendungen. Anschließend wird im vierten Kapitel die Realisierung und die Zusammenarbeit der verschiedenen Komponenten von Visual ScatterUnit eingehend erörtert. Das fünfte beschäftigt sich dann mit der automatisierten Erstellung von Softwarefamilien zur Anwendungsentwicklung mittels des modellgetriebenen Rahmenwerkes ScatterFactory. Hier wird die Integration von Visual Scatter Unit und ScatterFactory dargestellt, womit der Testprozess im Kontext der Anwendungsentwicklung in Betracht kommt.[4]

Der dritte Teil der Arbeit *WSN-Verwaltung* widmet sich in zwei Kapiteln der Verwaltung drahtloser Sensornetze und ihrer Plugin-orientierten Realisierung innerhalb der Werkzeugkette, durch welche die angestrebte Zusammenarbeit mit der im vorigen Teil behandelten generativen Infrastruktur erzielt werden soll. Gegenstand des sechsten Kapitels ist das ScatterPlug-Rahmenwerk, mit dessen Hilfe die Daten drahtloser Sensornetze sowie der auf ihnen laufenden Anwendungen visualisiert und statistisch analysiert werden können. Die Konfiguration, Überwachung und Softwareupdates des Sensornetzes bzw. der darauf laufenden Anwendungen wird mit Hilfe des ScatterEditors bewerkstelligt, welches im siebten Kapitel vorgestellt wird. Dabei wird erläutert, wie die Hauptmerkmale von ScatterEditor ebenfalls internetbasiert in Anspruch genommen werden können.

Im letzten Teil der Arbeit *Bewertung und Positionierung von ScatterClipse* wird in Kapitel 8 die Werkzeugkette ScatterClipse sowie deren Nutzen evaluiert, bevor in Kapitel 9 verwandte Arbeiten diskutiert werden. Abschließend wird in Kapitel 10 ein Fazit gezogen, wobei der integrierte Einsatz der Werkzeugkomponenten von ScatterClipse, der einen kohärenten Betrieb der Werkzeugkette bewirkt, anhand der *ScatterClipse AssemblyLine* geschildert wird. Vielfältige Beispiele zum Einsatz der Werkzeugkette finden sich im Benutzerhandbuch sowie in den Video-Demos auf der Webseite von ScatterClipse, welche unter [WebAlsa] zu finden ist.

---

[4] Auf die technische Realisierung von ScatterFactory wird nicht gesondert in einem eigenständigen Kapitel eingegangen, da diese analog zu Visual ScatterUnit vonstatten geht. Jedoch werden bestimmte technische Besonderheiten und Herausforderung innerhalb des Kapitels hervorgehoben und genauer untersucht.

2. Verwendete Technologien und Werkzeuge

Dieses Kapitel befasst sich mit den Technologien und Werkzeugen des modellgetriebenen Prozesses, wobei jedes Werkzeug für eine bestimmte Phase dieses Prozesses zuständig ist. Ferner wird die Integration der beschriebenen Werkzeuge behandelt, welche eine erfolgreiche Zusammenarbeit im Rahmen einer einzigen Werkzeugkette ermöglicht. Es geht speziell um diejenigen Werkzeuge, welche zur Realisierung der generativen Infrastruktur von ScatterClipse zum Einsatz kommen. Zunächst wird jedoch auf den architekturzentrierten Ansatz des modellgetriebenen Paradigmas eingegangen und auf die Frage, wie dieser sich von der modellgetriebenen Architektur abgrenzt.

2.1. Das architekturzentrierte modellgetriebene Paradigma[5]

Während die Objektorientierung gegenwärtig den Softwareentwicklungsprozess dominiert, hält ein neues Paradigma sowohl in der Industrie als auch in der Forschung Einzug. Der Ansatz, der in aller Munde ist, lautet modellgetriebene Softwareentwicklung[6] (MDSD) [WebMDSD]. Neu ist dabei, dass Modelle nicht nur zum Zweck der Dokumentation bzw. Visualisierung eingesetzt werden: Die semantikbehafteten und ausdrucksstarken formalen Modelle fungieren auch als Technik, um Schlüsselkonzepte und Randbedingungen der Domäne (Plattform) vollständig und prägnant zu repräsentieren. Um diesen Entwicklungsprozess für die Industrie einsatzfähig zu machen, wurde von der OMG[7] (eine Initiative von 800 Unternehmen) [WebOMG] die modellgetriebene Architektur[8] (MDA) [WebMDA] entwickelt. Abbildung 1 veranschaulicht die Abläufe des MDA-Prozesses sowie die zwischen ihnen bestehenden Abhängigkeiten:
Das Platform Independent Model (PIM) stellt die fachliche Spezifikation der darunterliegenden abzudeckenden Plattformen dar. Das Ziel ist dabei, eine universelle formale Modellierungssprache zur Verfügung zu stellen, die eine umfassende Abstraktion bewirkt und zwar von möglichst allen Schlüsselkonzepten der unterschiedlichen zu modellierenden Plattformen. Daher werden solche PIMs in einem ersten Transformationsschritt zu einem an die jeweilige Plattform angepassten Modell (Platform Specific Model, PSM) transformiert, woraus in einem weiteren Schritt die dieser Domäne korrespondierenden Codeartefakte generiert werden. Demzufolge wird dieser Vorgang als *Model to Model to Code* Transformation bezeichnet.

Wie bereits erwähnt, wird für die Realisierung der generativen Infrastruktur (ScatterFactory und Visual ScatterUnit) in ScatterClipse die architekturzentrierte Methode verfolgt, die eine Instanz bzw. Ausprägung des modellgetriebenen Paradigmas darstellt. Während die Zielvorgabe der OMG bezüglich der MDA die Steigerung der Portabilität und Interoperabilität der Software durch eine einheitliche und umfassende Modellierungstechnik ist, liegt der Fokus des modellgetriebenen architekturzentrierten Ansatzes in der Erhöhung der Effizienz und Robustheit des Softwareentwicklungsprozesses durch eine optimierte Automatisierung der Codeer-

[5] Architecture Centric–Model Driven Software Development: AC-MDSD.
[6] Siehe Glossar.
[7] Siehe Glossar.
[8] Siehe Glossar.

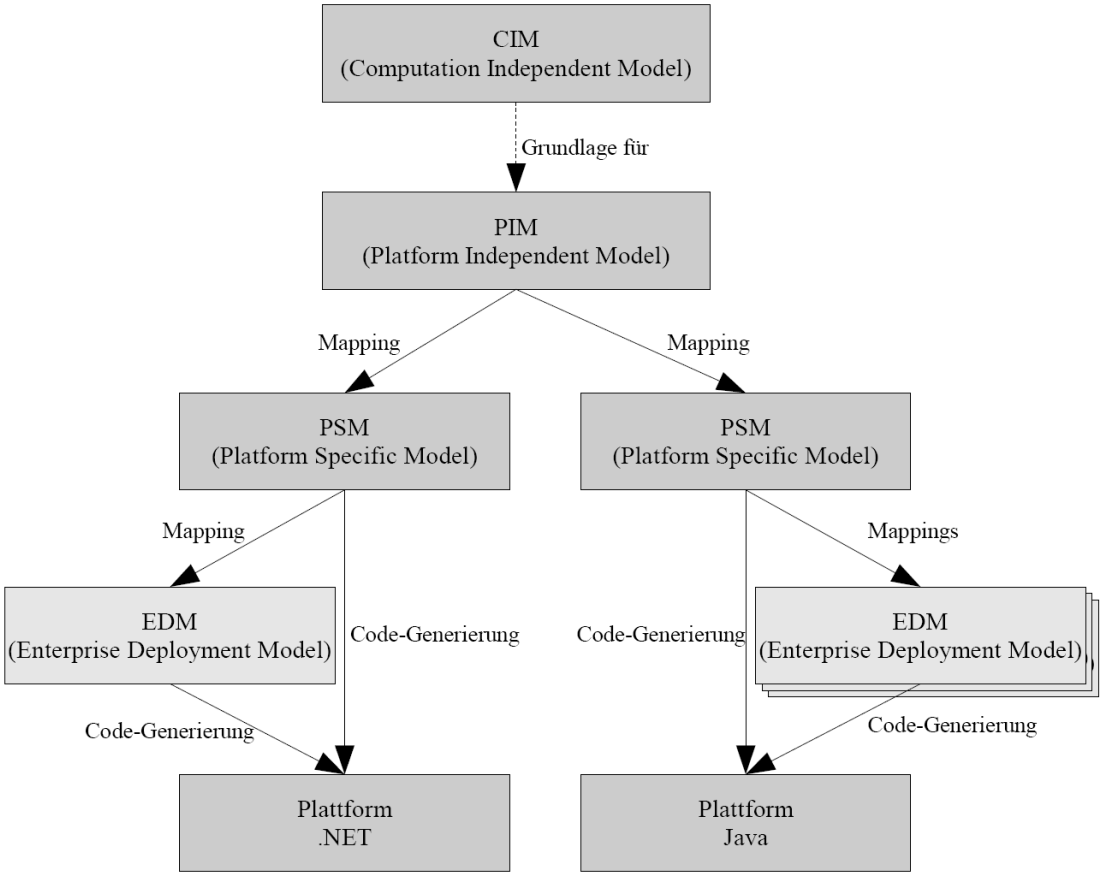


Abbildung 1 Der MDA-Prozess [Hent06]

stellung. Zugleich wird die Steigerung der Softwarequalität durch eine erhöhte Wiederverwendbarkeit der Codeartefakte angestrebt.

Daher liegt – wie der Name zum Ausdruck bringt – das Hauptaugenmerk auf der unterliegenden Domäne *Architektur*, für die gezielt modelliert wird. Folglich wird im Gegensatz zu MDA anstelle der Generierung derselben Softwareartefakte für unterschiedliche Plattformen die automatisierte Generierung vielfältiger Softwarevariationen (Softwarefamilien) einer bestimmten Domäne angestrebt. „Das Ziel architekturzentrierter, modellgetriebener Softwareentwicklung lautet also: Ganzheitliche Automation bei der Erstellung von Infrastrukturcode bzw. Minimierung von redundantem, technischem Code in der Anwendungsentwicklung.“ [Stah05]

Dieses Bestreben ergibt sich in erster Linie aus der Beobachtung, dass der oft mehrmalig und überlappend verwendete Infrastrukturcode einen beachtlichen Anteil am gesamten Code verschiedener Anwendungen derselben Domäne ausmacht. Während dieser Anteil in der E-Kommerz-Domäne bei etwa 70% liegt, steigt er bei der hardwarenahen Programmierung in der Domäne eingebetteter Systeme auf 90% bis 100%. [Czar00] Daher liegt es auf der Hand, die Erstellung gerade dieser Art von Code durch Codegeneratoren zu automatisieren, so dass der Entwickler sich auf den eigentlichen Problembereich, nämlich die Anwendungslogik, konzentrieren kann. Dadurch richtet sich der Fokus des Entwicklers auf die Gegebenheiten der Domäne und kann sich von den damit verbundenen vielseitigen technischen Belangen lösen.

Deren Realisierung wird durch den Infrastrukturcode bewerkstelligt, dessen Erstellung an den Codegenerator delegiert wird. Die Voraussetzung dafür ist eine balancierte Abstraktion von der unterliegenden Domäne, die dem Entwickler die Konfrontation mit den technischen Belangen der Domäne erspart, die architektonischen Randbedingungen der Domäne jedoch erfasst. Dies verlangt eine adäquate Modellierung, bei der die Domäne im Mittelpunkt steht. Im Kontext der AC-MDSD erfolgt dies durch den Entwurf einer entsprechenden domänenspezifischen Modellierungssprache, Domain Specific Language[9] (DSL), die eine domänennahe Modellierung ermöglicht.

Solche domänenspezifischen Modellierungssprachen haben gegenüber den in der MDA [Petr06] verwendeten UML-Modellen [Rupp05] den Vorteil, dass sie sich von Grund auf an die bereits vorhandene Domäne anlehnen, da deren Entwurf und Aufbau maßgeschneidert auf die Randbedingungen der Domäne ausgerichtet sind. Damit kann man auf einfache Weise die verschiedenen Aspekte der Domäne in korrespondierenden Modellen nahtlos und präzise erfassen. So gewähren sie einen hohen Abstraktionsgrad und bleiben zugleich bezüglich der Domäne konkret. Anders ist es bei den schnell komplex werdenden UML-Modellen, die auf Grund ihrer eher universellen Motivation den Entwickler relativ schnell in die Lage bringen, sich mit mehrseitigen Modellkonstrukten simultan auseinander setzen zu müssen, was die Modellierungskomplexität deutlich erhöht.

Aufgrund ihrer universellen Natur müssen UML-Modelle nachträglich der unterliegenden Domäne leichtgewichtig und somit bedingt angepasst werden, was der angestrebten domänennahen Modellierung entgegenwirkt. Die dadurch verminderte Ausdruckkraft hinsichtlich der Domäne weitet sich zu einer semantischen Lücke zwischen der Modellierungssprache und den daraus zu generierenden Codeartefakten aus. Dies hat zur Folge, dass im Kontext der MDA der Transformationsprozess keinen direkten Modell-zu-Code, sondern eine *Modell-zu-Modell-zu-Code*-Überführung durchlaufen muss, um die entstandene semantische Lücke zu überwinden. Dieser Sachverhalt führt nicht nur zu einem schwerer zu beherrschenden Modellierungsmechanismus, sondern steigert die Komplexität des Prozesses zur automatisierten Gewinnung der Domänensoftwareartefakte aus den abstrakten Modellen. [Evan03]

Insgesamt kann die Zielsetzung des AC-MDSD-Paradigmas mit der Verwendung von modernen Produktlinien in der Automobilindustrie verglichen werden (vgl. Abbildung 2). Am Anfang steht der Prototyp *(Reference Implementation)*, der die wichtigsten Konzepte beinhaltet. Der Prototyp zeigt, wie das zu produzierende Fahrzeug aussehen soll. Der Konstruktionsplan (*Modell*) dient als Ausgangsbasis für das Endprodukt (*generiertes Artefakt*) und gibt an, welche Einheiten (*Komponenten*) notwendig sind. Um die Konstruktion der Produktlinie (*generative Architektur*) ebenso wie die spätere Produktion (*Codegenerierung*) zu vereinfachen, werden logisch zusammengehörende Komponenten zu Produktionseinheiten zusammen gefasst. Produktionseinheiten, die nicht automatisiert werden oder für die Automatisierung zu kompliziert sind, werden in Handarbeit entwickelt (*manueller Code*). Um eine breite Produktpalette anzubieten (*Systemfamilie*) werden die Komponenten im Regelfall während des Produktionsprozesses variiert, während die Produktionsplattform als solche unverändert bleibt. [Hent06]

[9] Siehe Glossar.

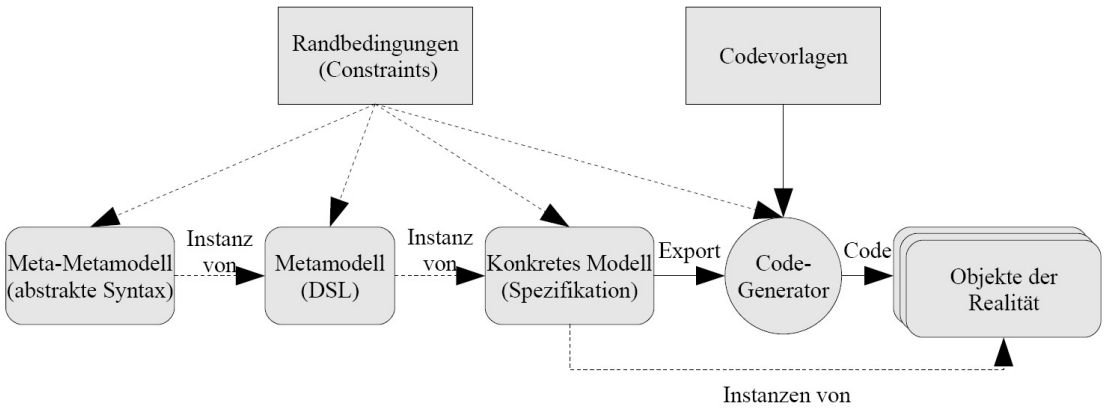


Abbildung 2 Der AC-MDSD-Prozess [Hent06]

Bevor in den nächsten beiden Kapiteln die Architektur der architekturzentrierten generativen Infrastruktur (Visual ScatterUnit und ScatterFactory) sowie die Zusammenarbeit der verschiedenen Architekturkomponenten behandelt werden, ist es an dieser Stelle unerlässlich, sich einen Einblick in die Werkzeuge zu verschaffen, aus denen die generative Infrastruktur zusammengebaut ist.

Wie eingangs erwähnt, bildet die modellgetriebene generative Infrastruktur architektonisch betrachtet, eine Plugin-orientierte Werkzeugkette, wobei die Eclipse Plattform als deren Basis fungiert. Jede Phase bzw. jeder Schritt des modellgetriebenen Softwareentwicklungsprozesses wird von einem Werkzeug der Werkzeugkette ausgeführt, wobei das Resultat eines Werkzeuges oft als Eingabe des darauf folgenden fungiert. Auf Basis des mit dem EMF erstellten Metamodells wird ein domänenspezifischer grafischer Editor mit Hilfe von GMF erzeugt. Die durch den Editor modellierten Konstrukte werden von openArchitectureWare eingelesen, validiert und im Anschluss auf deren korrespondierenden Templates abgebildet, um dann den Codegenerierungsprozess anzustoßen. Mit Hilfe des CDT [WebCDT] und des Ant Plugins [WebANT] von Eclipse werden dann die generierten Softwareartefakte kompiliert, um ein Code Image zu erzeugen.

Es ist daher nicht verwunderlich, dass die für die Konstruktion der Werkzeugkette verwendeten Rahmenwerke ausschließlich Eclipse-Projekte sind. Dies ermöglicht eine optimale Integration der Werkzeuge, wodurch eine einwandfreie Zusammenarbeit gemäß dem modellgetriebenen Paradigma erzielt wird. Alle verwendeten Rahmenwerke sind Opensource-Projekte.

2.2. Die Eclipse Plattform

Was ist Eclipse? Antwortet man auf diese Frage, dass Eclipse lediglich eine Entwicklungsumgebung für Java ist, so hat man – spätestens nach der Entwicklung von Eclipse 3.0 – eine falsche Antwort gegeben. Es muss an dieser Stelle betont werden, dass Eclipse – trotz eines hohen Marktanteils für die Entwicklung mit Java – eine allgemeine, umfassende und betriebssystemunabhängige Plattform darstellt, die nicht auf ein spezielles Gebiet beschränkt ist: „alles mögliche und nichts im besonderen“. [Daum06a] Dass Eclipse nicht auf Java beschränkt ist, wird sich auch in dieser Arbeit deutlich zeigen, da überwiegend C-Code, Makefiles und andere Templates- bzw. Spezifikationssprachen für die Codegenerierung bzw. Live-Modellvalidierung unter Eclipse mit Hilfe von diversen speziellen Wizards und Plugins bearbeitet bzw. erstellt werden.

Eclipse bildet einen Kern, an den für unterschiedliche Zwecke und unabhängig voneinander entwickelte Plugins angeschlossen und integriert werden. Dies entspricht dem Open Service Gateway Initiative Standard[10] (OSGi). Dabei fungiert der Eclipse-Kern als OSGi-Server [WebOSGI], bei dem sich die jeweiligen Klienten (Plugins) eindeutig registrieren müssen (vgl. Abbildung 3). Diese Eindeutigkeit spielt eine entscheidende Rolle, denn so ist es beispielsweise möglich, dass verschiedene Versionen desselben Plugins gleichzeitig in Anspruch genommen werden können. Um die Leistungsfähigkeit zu optimieren, werden die Plugins erst dann aktiviert, wenn sie betätigt werden. Jedoch müssen alle Plugins (Klienten) beim Start registriert bzw. erfasst werden, was eine längere Startphase der Eclipse-Plattform verursacht. [Daum06a]

Um eine koordinierte und softwaretechnisch konforme Zusammenarbeit der oft zeitlich unabhängig voneinander entwickelten Plugins zu gewährleisten, kommunizieren diese mittels spezifischer XML-basierter Schnittstellen namens Erweiterungspunkte (Extension Points). Ein Plugin kann bereits vorhandene Erweiterungspunkte entsprechend erweitern bzw. sich an sie anpassen oder eigene Erweiterungspunkte definieren, um diese dann von anderen Plugins erweitern zu lassen. Für eine standardisierte Entwicklung von eigenen Plugins bietet Eclipse nützliche Funktionalitäten mit dem *Plugin Development Environment* (PDE) an.

Es ist möglich, die Plugins in verschieden Ausprägungen zu realisieren. Neben den herkömmlichen Plugins können sie als *Actions* und *Views* implementiert werden. *Actions* werden oft als Steuerungselemente in ein Haupt-Plugin eingebettet und dann vom Benutzer ab und an betätigt. Sie bieten unterstützende Funktionalitäten, die oft vom Benutzer nicht unbedingt längerfristig in Anspruch

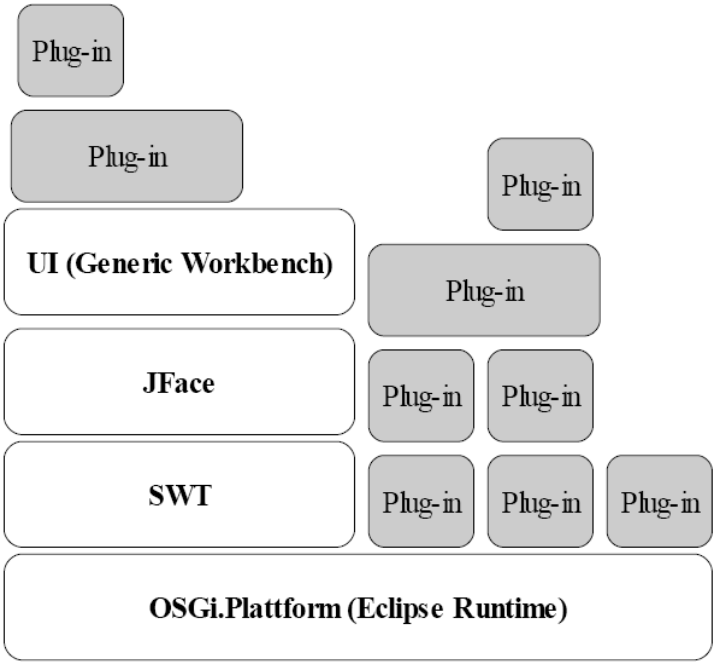


Abbildung 3 Die Eclipse-Plattform als OSGi-Server [Hent06]

[10] Die aktuelle geschützte Bezeichnung lautet *OSGi Alliance*.

genommen werden. Daher wurden beispielsweise in ScatterPlug (vgl. Kapitel 6) bestimmte Funktionalitäten wie der Verbindungsaufbau und -abbruch zum Sensornetz sowie das Scannen des Netzes zur Prüfung der platzierten Sensoren als Actions realisiert.

Views sind Fenster, die spezifische Funktionen zur Verfügung stellen. Der Benutzer kann diese je nach Bedarf aktivieren und individuell in der Umgebung platzieren. Damit bietet das Konzept der Views in Eclipse ein bemerkenswertes Merkmal, um eine optimale ergonomische Interaktion mit dem Benutzer zur erzielen. Von diesem Konzept haben wir insbesondere im Entwurf des Verwaltungs- und Überwachungspfads der Sensorknoten ausgiebig Gebrauch gemacht. Hier wurden verschiedene, mit dem Benutzer interagierende und illustrative Funktionalitäten als Views realisiert, wie die Visualisierung, die Statistik und der gesamte ScatterEditor. Dies sehen wir als wichtigstes Merkmal des Entwurfes der WSN-Verwaltung bei ScatterClipse an, um den Anforderungen an die in der Forschung diskutierte Mensch-Maschine-Schnittstelle (*human-computer interfaces*) gerecht zu werden.

Für die Entwicklung von grafischen Benutzerschnittstellen (GUIs) bietet Eclipse mit Standard Widgets Toolkit (SWT) und JFace ein breites Spektrum an Bibliotheken, vergleichbar der von Swing im JDK. Das Hauptaugenmerk liegt dabei auf dem *look and feel*-Prinzip. Das Ziel ist eine betriebssystemunabhängige grafische Programmierung, wobei die Ansicht der Benutzerschnittstellen sich dynamisch dem darunter liegenden Betriebssystem anpasst. Zu diesem Zweck wird für das jeweilige Betriebssystem eine Wrapper-Klasse verwendet. Dies führt jedoch zu einem Leistungsverlust, insbesondere bei anspruchsvollen Grafikanwendungen unter Eclipse. Eine Diskussion der Vor- und Nachteile von SWT und JFace findet man in [Daum06a, S. 154-156]. Für anspruchsvolle grafische Programmierung ist u.a. auf [Daum06b, Kapitel 1 und 2] zu verweisen.

2.3. Das Eclipse Modeling (EMF)

Wie aus der bisherigen Schilderung deutlich wurde, ist die Konstruktion domänenspezifischer Modellierungssprachen eine anspruchsvolle Tätigkeit. Damit dieser Prozess mit der nötigen Effizienz vonstatten geht, sind angemessene Metamodellierungsrahmenwerke unerlässlich [WebBart]. Das Eclipse Modeling ist ein solches Rahmenwerk. Das Hauptziel des von IBM entwickelten Rahmenwerks ist dabei, auf Basis des von der OMG spezifizierten *Meta modeling Objects*[11] (MOF) [WebMOF] Standard Modelle zu formulieren, mit deren Hilfe editierbare domänenspezifische Sprachen definiert werden können. Editierbar bedeutet in diesem Zusammenhang, dass die der DSL zugehörigen Editoren automatisiert und gemäß dem definierten Metamodell erzeugt werden. [Moor04] Zu diesem Zweck stellt EMF Ecore als Beschreibungssprache zur Erstellung von Metamodellen zur Verfügung, eine mit XML Model Interchange[12] (XMI) [WebXMI] serialisierbare Java-Implementierung des MOF-Kernes (Essential MOF, EMOF). Die Präsentation von mit Ecore beschriebenen (Meta)Modellen erfolgt durch strukturelle Formalisierungen wie XML-Schemata oder UML-Notation. Abbildung 4 zeigt eine UML-orientierte Übersicht der Elemente von Ecore und deren Relationen:

[11] Siehe Glossar.
[12] Siehe Glossar.

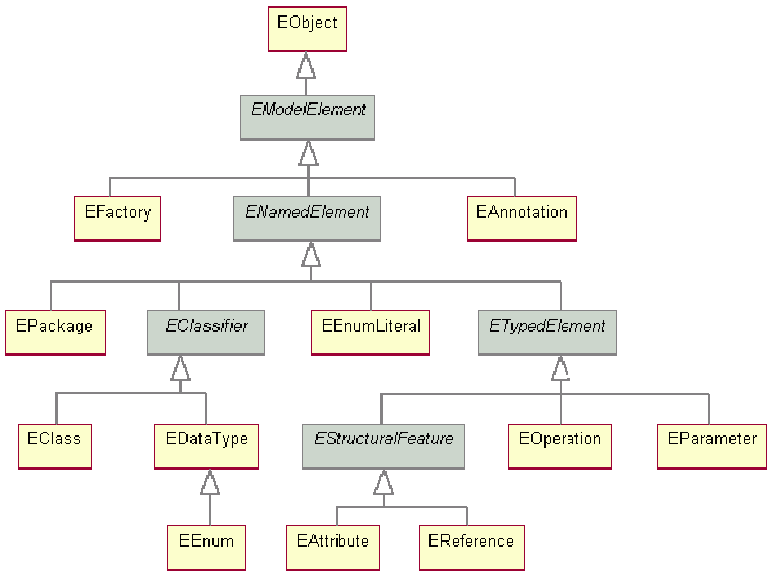


Abbildung 4 Die Elemente von Ecore als Klassenhierarchie [WebEcore]

Man kann die Klasse *EObject* mit der Objekt-Klasse in Java vergleichen. Sie bildet die Basisklasse, von der die restlichen Klassen erben bzw. die diese erweitern. Wie die Erfahrung mit der Modellierung mit Hilfe von UML zeigt, bestehen die meisten (statischen) Modelle überwiegend aus den folgenden Grundelementen: Klassen, Assoziationen, Attribute, Aufzählungstypen und Typisierungen der Attribute. Dies ist ähnlich bei Ecore. Die gerade erwähnten Grundelemente entsprechen in der Ecore-Klassenhierarchie den folgenden Klassen (der Reihe nach): *EClass*, *EReference*, *EAttribute*, *EEnum* und *EDataType*.

Hiermit stehen die erforderlichen Schlüsselkonzepte zur Verfügung, um die abstrakte Syntax von eigens erstellten domänenspezifischen Sprachen festzulegen. Zur Bearbeitung von Ecore-Modellen liefert EMF den entsprechenden Editor. An dieser Stelle stellt sich die Frage, wie man der durch EMF definierten abstrakten Syntax eine korrespondierende statische Semantik verleihen kann.

Auf der bislang betrachteten Ebene ist es durch EMF möglich, die auf Basis von Ecore erstellten Modelle begrenzt auf deren semantische Korrektheit zu prüfen. Daher werden lediglich z.B. die Korrektheit der Attributierung der Klassen bzw. die Endungen der Assoziationen überprüft. Diese Beschränkung ist jedoch bewusst im Entwurf von EMF enthalten, um die Komplexität der anspruchsvollen Tätigkeit der Metamodellierung zu reduzieren. Weitere, fortgeschrittene Semantikdefinitionen werden auf die Modellierungsebene verlagert, für die eine Vielzahl von Wizards und Hilfswerkzeugen für diverse Aufgaben vorhanden ist. Andere komplexe Randbedingungen, die spezifische Eigenschaften und Gegebenheiten der darunter liegenden Domäne prägnant repräsentieren, werden weiter auf die untere Generierungsebene delegiert. Solche spezifischen, domänennahen Randbedingungen können dann beispielsweise im Kontext der Codegenerierung in Betracht gezogen werden, was den Codegenerierungsprozess optimiert.

Zusammenfassend kann man sagen, dass sich die Semantikdefinition auf drei Ebenen erstreckt, nämlich die Metamodellebene, die (Modellierungs-)Editorebene und die Generatorebene. Auf der Metamodellebene wird das Gerüst der DSL festgelegt,

auf der Modellierungsebene werden die Feinheiten des Editierens der Modellkonstrukte behandelt und auf der Generatorebene wird die DSL gegen domänenspezifische Eigenschaften validiert. Diese schrittweise Unterteilung reduziert zum einen die Komplexität, und zum anderen findet dadurch eine klare Teilung von Belangen statt.

Bevor wir tiefer auf die generative Schicht von EMF eingehen, bleibt zu erwähnen, dass EMF eine Option anbietet, in XML oder UML präsentierte Modelle zu importieren und nach deren Umwandlung weiter zu bearbeiten. Dies wird durch die Fähigkeit ermöglicht, Modelle in XMI als Austauschformat zu serialisieren. Es ist daher nicht verwunderlich, dass an der XMI-Standardisierung und Optimierung in Forschung und Industrie fortwährend gearbeitet wird. [WebJeck] Sie erhöht den Grad der Interoperabilität der verschiedenen MDSD-Werkzeuge, wodurch deren Integration deutlich erleichtert wird. Dies stellt einen Meilenstein für die Konstruktion von für den MDSD-Prozess tauglichen Werkzeugketten dar.[13]

Abbildung 5 zeigt die Komponenten der generativen Ebene von EMF sowie deren Zusammenarbeit. Das Generatormodell ist die Ausgabe der Transformation eines Wizards, dessen Eingabe ein Ecore konformes Modell ist. Zusammen mit den Java Emitter Templates fungiert das nun bearbeitbare Generatormodell als Eingabe für die EMF-Generatorkomponente. Die Templates steuern die Codegenerierung und bieten die Möglichkeit, die Artefakte des Generatormodells mit Templates zu verknüpfen. [Hent06]

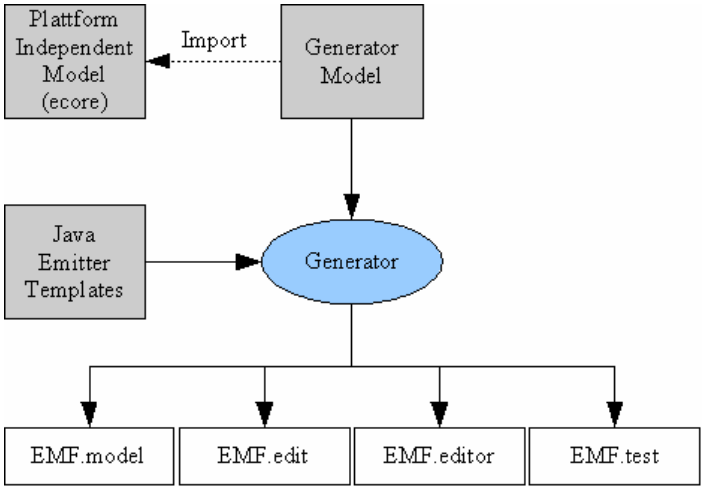


Abbildung 5 Die generative Schicht von EMF [Hent06]

Das Resultat dieses Generierungsprozesses sind die für die weiteren Schritte nötigen Artefakte. Das EMF.model liefert eine programmierbare Javacodierung des ursprünglichen Ecore-Models. Für die Erstellung des generativen, aus dem Metamodell gewonnen Editors sind EMF.edit und EMF.editor zuständig. An dieser Stelle ist anzumerken, dass die Eclipse Community bestrebt ist, sowohl den gerade erwähnten generativen Editor als auch die Java Emitter Templates standardmäßig durch andere, umfangreichere Technologien bzw. Werkzeuge zu ersetzen. So werden die

[13] Diese erfolgreiche Interoperabilität wird durch die Werkzeugkette von ScatterClipse anschaulich illustriert.

JETs durch die weitaus mächtigere oAW-Engine und der EMF-Editor durch das Graphical Modeling (GMF) abgelöst. Dies entspricht auch der Zusammensetzung der Werkzeuge der generativen Infrastruktur von ScatterClipse.

2.4. Graphical Modeling (GMF)

Wurde das Metamodell einer DSL definiert, wäre es optimal, einen auf das Metamodell zugeschnitenen Editor für die DSL automatisiert zu erzeugen. Dies bewerkstelligt das im Juni 2006 veröffentlichte *Graphical Modeling* (GMF), das mit dem im letzten Abschnitt behandelten EMF zusammenarbeitet.

Die enge Zusammenarbeit wird in erster Linie vom Common Layer der Runtime Component im GMF abgewickelt. Dies wird durch die Fähigkeit der Runtime Component ermöglicht, die entsprechenden Dienste vom EMF in Anspruch zu nehmen und diese in Verbindung mit den eigenen Funktionalitäten einwandfrei auszuführen. Die erfolgreiche Zusammenarbeit zwischen den beiden Werkzeugen zeigt sich insbesondere an der Unterstützung der grafischen Validierung von Ecore-basierten Modellen im GMF Editor, wodurch die Modellintegrität gewährleist wird. Um die auf Basis des Ecores definierten Modelle mit einer ausdrucksstarken Semantik zu formalisieren, ermöglicht es GMF, domänenorientierte Randbedingungen an die jeweiligen Modelle zu binden. Hierzu werden die Randbedingungen mit Hilfe von Ausdrücken der *Object Constraint Language*[14] (OCL) [WebOCL] spezifiziert. Die OCL-Ausdrücke werden dann mit Hilfe eines zugehörigen Wizards an das jeweilige Ecore-Modell gebunden.

Zu diesem Zweck bedient sich die Runtime Component der *Eclipse Modeling Technology* (EMFT). Mit dieser werden die OCL-Randbedingungen, die an den GMF-Editor gebunden sind, eingelesen und ihre Einhaltung im Modell überwacht. Verstößt man während der Modellierung gegen bestimmte Randbedingungen, werden die entsprechenden Fehlermeldungen angezeigt. Diese sind sowohl visuell als auch verbal: Die visuellen Fehlermeldungen werden im Editor bei den betreffenden Modellkonstrukten eingeblendet, die verbalen sind die vom Benutzer bzw. Domänenexperten selbst spezifizierten Fehlermeldungen. Diese werden dank der Eclipse-Integration im *Problem View* angezeigt. Modellinstanzen, die gemäß dem Ecore-Metamodell nicht verknüpft werden können, lassen sich auch im Editor nicht verbinden. Der folgende Bildschirmabzug von ScatterFactory in Abbildung 6 illustriert dieses Vorgehen. [Hent06]

Gemäß der EMFT werden die üblicherweise vom DSL-Designer[15] in OCL spezifizierten Bedingungen an einer zentralen Stelle gehalten, dem Audit Container. Mit Hilfe der zugehörigen Werkzeuge werden auf einfache Weise die Bedingungen deren jeweilig korrespondierenden Modellkonstrukten zugewiesen. Dabei ist es zum Einen möglich, den entsprechenden Kommentar der im Problem View angezeigten Fehlermeldung festzulegen, und zum Anderen wird die Art der visuell im DSL-Editor

---

[14] Siehe Glossar.

[15] Oft handelt es sich in diesem Zusammenhang um den Domänenexperten. Die DSL wird allerdings nicht in einem Guss entworfen, sondern mehr oder weniger iterativ nach dem Wasserfallprinzip konstruiert. Hier zeigt sich die Stärke der harmonischen Integration der beteiligten Rahmenwerke der Werkzeugkette. Ein Beispiel dafür ist die Möglichkeit, den generativen DSL-Editor (GMF) nach jeder Verfeinerung des Metamodells (EMF) automatisiert zu generieren.

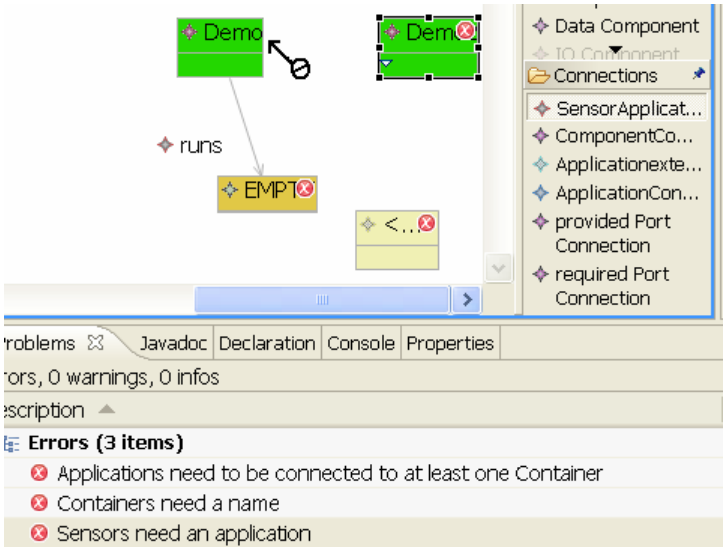


Abbildung 6 Modell-Validierung im GMF Editor [Hent06]

eingeblendeten Fehleranzeige angegeben (z.B. ob es sich um einen Modellierungsfehler, eine Warnung oder einen Tipp handelt).

Bislang wurde noch nicht über die Erstellung des generativen Editors in GMF gesprochen. Dieser Vorgang beruht wiederum auf dem Graphical Editing. Die Realisierung des Editors nur mit Hilfe von GEF gestaltet sich sehr umständlich, da auf Grund des importierten Ecore-Metamodells die notwendigen abstrakten Klassen erzeugt werden, die dann vom Entwickler manuell implementiert werden müssen. Es ist daher naheliegend, diesen Prozess zu automatisieren. Diese Aufgabe erledigt GMF und zwar modellgetrieben. Dabei werden aus dem aus EMF importierten Metamodell verschiedene Modelle automatisiert generiert, die dann als Eingabe für weitere Wizards und Werkzeuge fungieren. Dabei ist jedes Modell für die Realisierung einer bestimmten Aufgabe zuständig. Somit findet eine Teilung von Belangen statt, was die Steuerung der automatisierten Generierung des Editors strukturiert und ihre Komplexität deutlich verringert.

Abbildung 7 veranschaulicht die verschiedenen generierten Modelle sowie deren Abhängigkeiten und den Ablauf ihrer Zusammenarbeit bis hin zur Editorerzeugung. Im Folgenden sind die verschiedenen Modelle sowie deren Zuständigkeiten aufgelistet: [Hent06]

- Das *Graphical Definition Model* (.gmfgraph): Hier wird festgelegt, wie die einzelnen Elemente des Metamodells visuell im Editor durch ihre entsprechenden Figuren erscheinen. Dieses Vorgehen wird vom Benutzer durch die entsprechenden Wizards bestimmt.
- Das *Tooling Definition Model* (.gmftool): Die Vorgehensweise ist analog zum obigen Modell, allerdings werden hier die einzelnen Elemente der Palette (vgl. Abbildung 6, rechts vom Editor) bearbeitet. Neben der Konfiguration der visuellen Erscheinung können die Elemente der Palette gemäß dem Design der DSL zu einer Kategorie zusammengefügt werden. Ein Beispiel ist die Erstellung der Kategorie der Modellinstanzen *Connections* und die Einordnung der verschiedenen *Connections* Ausprägungen.

- Das *Mapping Model* (.gmfmap) und das *Generator Model* (.gmfgen): .gmfgraph und .gmftool fungieren als Eingabe für das *Mapping Model*, das diese dann unter Betrachtung der vom Benutzer im Audit Container spezifizierten Semantik zusammenfügt. Das Resultat ist das vom Benutzer zwecks Editoreinstellung individuell konfigurierbare *Generator Model*, aus dem dann unmittelbar der generative Editor entsteht.

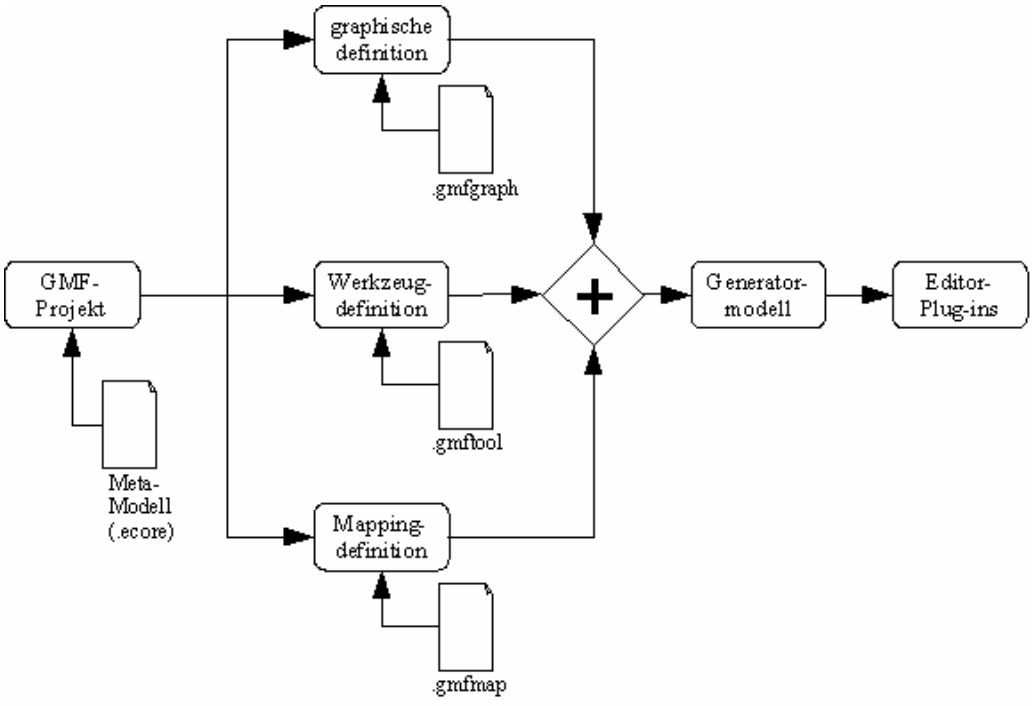


Abbildung 7 Die automatisierte Erstellung des Editors im GMF [Hent06]

2.5. openArchitectureWare (oAW)

Das Hauptaugenmerk für die Erzielung der angestrebten Vorteile im modellgetriebenen Softwareentwicklungsprozess liegt auf der Durchführung der entsprechenden Automatisierung. Dafür ausschlaggebend ist eine optimierte Codegenerierung der Softwareartefakte, welche durch die DSL modelliert werden. Das Vorhandensein einer solchen Komponente zur automatisierten Codegenerierung ist für den MDSD-Prozess daher unerlässlich.

Das von b+m Informatik AG [Web b+m] entwickelte und im Jahr 2003 veröffentlichte opensource Rahmenwerk ist eine solche Komponente: „Das Hauptziel von openArchitectureWare ist es, eine äußerst flexible Plattform für MDSD-Prozesse zu bieten: Es versteht sich als Template-basiertes Rahmenwerk zur Erstellung eigener Generatoren nach dem Model-to-Code-Ansatz auf der Basis beliebiger Modelle.“ [Petr06, S. 191]

Ein prägnantes Merkmal von oAW ist demnach der *Model-to-Code-Ansatz*, was unserer in ScatterClipse verfolgten architekturzentrierten Methode entgegenkommt, wie zu Beginn des Kapitels dargestellt wurde. Abbildung 8 zeigt die Architektur von oAW und deren Komponenten:

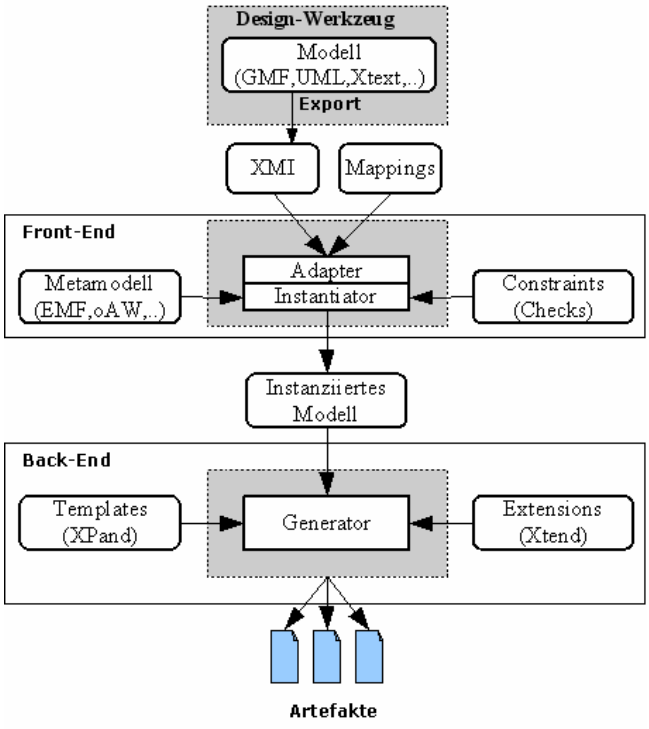


Abbildung 8 Die Architektur und Technologien von oAW [Hent06]

Im Folgenden werden die Funktionalitäten der verwendeten Technologien und die Komponenten der Architektur sowie deren Zusammenarbeit erläutert.

Zuerst muss das formale Modell, aus dem die korrespondierenden Softwareartefakte gewonnen werden, vom Codegenerator-Rahmenwerk korrekt eingelesen werden, um dieses Modell (Eingabe) den übrigen Komponenten des Codegenerator-Rahmenwerks zur Weiterverarbeitung zur Verfügung zu stellen. Zu diesem Zweck wird das üblicherweise in XMI[16] serialisierte Modell importiert und dann vom Parser als Syntaxbaum zur Verfügung gestellt.

Der Syntaxbaum ist als Objektgeflecht repräsentiert, was im Java-basierten oAW bedeutet, dass die Knoten des Syntaxbaumes als Java-Klassen repräsentiert werden. Diese sind gemäß dem Metamodell (abstrakte Syntax des Modells) definiert, auf dessen Basis das eingegebene formale Modell gebildet wurde. Daher muss das Metamodell ebenfalls als Java-Repräsentation vorliegen, um vom oAW-Rahmenwerk (zwecks Validierung des eingegebenen formalen Modells) bearbeitbar und interpretierbar zu sein.

Welches Metamodell das Rahmenwerk dabei verwendet, ist vom Benutzer frei konfigurierbar. Hierzu unterstützt oAW standardmäßig Ecore-konforme Metamodelle sowie das Metamodell für UML 2. Da leider das XMI der verschiedenen UML-Werkzeuge (noch) nicht einheitlich ist, stellt oAW eine Reihe von entsprechenden Adapterklassen für die jeweiligen Werkzeuge zur Verfügung. [Efft08] Dadurch wird eine breite Palette von UML-Werkzeugen unterstützt. Um nun aus dem eingegebenen Metamodell die äquivalente Java-Repräsentation zu erzeugen, stellt oAW einen

[16] Da möglicherweise andere Austauschformate existieren, werden von oAW dafür zuständige (benutzerdefinierte) XML-basierte Mapping-Dateien zur Verfügung gestellt.

speziellen Generator zur Verfügung. Handelt es sich dabei (wie es oft der Fall ist) um ein Ecore-Metamodell, so ist keine Metamodelltransformation mittels des Generators in das äquivalente Java-Objektgeflecht vonnöten, da auf die bereits in EMF generierte Java-Repräsentation des Metamodells zugegriffen werden kann. Hier sieht man ein weiteres Beispiel für die enge und sich ergänzende Zusammenarbeit der verwendeten Rahmenwerke in der Werkzeugkette von ScatterClipse.

Der Syntaxbaum des eingelesenen formalen Modells liegt, wie erwähnt, als bearbeitbares Objektgeflecht von Java-Instanzen vor. Daher spricht man in diesem Zusammenhang von einem instanziierten Modell. Bevor der eigentliche Codegenerierungsprozess angestoßen wird, wird die Integrität der Objekte des gerade instanziierten Modells verifiziert. Die Modellintegrität ist erhalten, wenn der Syntaxbaum des formalen Modells tatsächlich eine Instanz des Metamodells darstellt. Zusätzlich muss die zugehörige formalisierte Semantik erfüllt sein.

Diese Überprüfung geschieht mit Hilfe der von oAW gelieferten Sprache Checks[17] [WebCheck]. Die Checks-Ausdrücke, mit denen die Semantik des Metamodells formuliert wird, werden in einer zentralen Datei „*.chk“ gehalten. Damit können diese einerseits für verschiedene Metamodelle derselben Domäne wiederverwendet werden, andererseits können sie leichter vom Benutzer verwaltet und von der Architektur bearbeitet werden. Um die Modellintegrität (Semantiküberprüfung) bezüglich der Checks-Ausdrücke zu überprüfen, müssen die Objekte des instanziierten Modells diesen zentral gehaltenen Randbedingungen zugänglich gemacht werden. Die Checks-Ausdrücke müssen in der Lage sein, auf die Methoden und Attribute des Objektgeflechtes des Modells zuzugreifen.[18] Dies soll anhand des Beispiels eines einfachen Ecore-konformen Modells demonstriert werden. Das folgende Diagramm in Abbildung 9 zeigt das Beispielmodell in der UML-Notation. [Hent06]

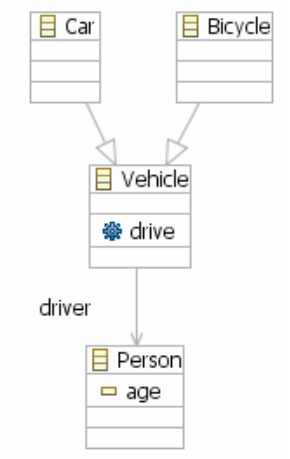


Abbildung 9 Ein einfaches mittels Checks zu validierendes Modell [Hent06]

[17] oAW bietet auch den dazu gehörigen Editor, der z.B. Codevervollständigung und Hervorheben von Syntaxelementen unterstützt. Dadurch können die mit Checks formulierten Randbedingungen robuster geschrieben werden.

[18] Hier sieht man ein Beispiel für den Nutzen, den die vorliegende Java-Repräsentation des betreffenden Modells mit sich bringt.

Anhand der im Ecore-Modell standardmäßig unterstützten internen Referenzen bzw. Collection kann auf wichtige Informationen der Ecore-Objekte zugegriffen werden, wodurch deren Kontext bezüglich der definierten Checks-Randbedingungen bestimmt wird. So kann mittels der Referenz eContainer eines Ecore-Objektes dessen Vaterobjekt im Modellbaum ermittelt werden. Durch die Collection eContents können sämtliche Kinderobjekte eines Objektes geliefert werden.

Mit diesem Mechanismus kann man auf einfache Art und Weise den gesamten Modellbaum traversieren und währenddessen die jeweiligen Objekte und die zwischen ihnen im Modell sich befindenden Assoziationen bearbeiten und überprüfen. Das folgende Beispiel in Programmauszug 1 zeigt einen einfachen Check-Ausdruck, der an das in Abbildung 9 gezeigte Modell gebunden ist. Dabei wird illustriert, wie z.B. vom eContainer zwecks Modellvalidierung Gebrauch gemacht wird.

Import Example[19];

Context Bicycle **ERROR** „Fahrräder müssen einen Fahrer haben“[20];

Bicycle.eContents.driver <> Null;

Programmauszug 1 Ein einfacher Checks-Ausdruck [Hent06]

Innovativ ist die Unterstützung von just-in-time-Validierung bzw. Live-Validierung des Modells anhand der daran gebundenen Checks-Randbedingungen. Dies wird durch den GMF-Adapter ermöglicht, den oAW seit der Version 4.1 anbietet. Dabei ist es möglich, durch einen *Watchdog* jegliche Änderung der Modellkonstrukte im Editor zu überwachen und währenddessen in Abhängigkeit der vollzogenen Modelländerungen die Einhaltung der zentral gehaltenen Checks-Randbedingungen zu überprüfen. Abbildung 10 zeigt die bei der Live-Validierung beteiligten Architekturkomponenten:

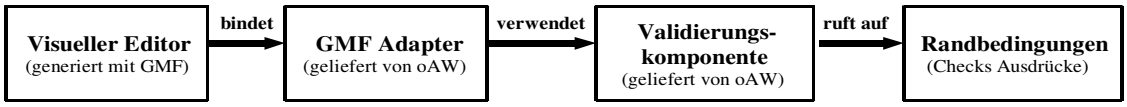


Abbildung 10 Die Architektur der Live-Validierung

Nun kann die Codegenerierung angestoßen werden, indem aus den Templates jeweils der für die Modellkonstrukte korrespondierende Code emitiert wird. Als Template-Sprache wird in oAW Xpand[21] [WebXpand] eingesetzt, eine einfache deklarative Skript-Sprache. Der Inhalt der Templates kann beliebig sein. So können unterschiedliche Softwareartefakte automatisiert aus den Templates generiert werden, variierend vom Code hoher Programmiersprachen bis zu Makefiles und Konfigurationsdateien.

---

[19] Durch die Import-Anweisung kann das betreffende Metamodell importiert werden, das im obigen Beispiel in Abbildung 9 *Example* heißt.

[20] Dies ist die Fehlermeldung, die im GMF-Editor visuell während der Live-Validierung angezeigt wird. Dies ist möglich dank des GMF-Adapters, der seit oAW Version 4.1 unterstützt wird.

[21] Analog zu Checks bietet oAW einen gesonderten Editor für Xpand mit Codevervollständigung und dem Hervorheben von Syntaxelementen.

Die Sprache bietet verschiedene Funktionalitäten um den Generierungsprozess zu optimieren. Ein Feature wurde jedoch absichtlich ausgelassen: die Bestimmung der Terminierung des Generierungsprozesses, die beim Entwurf von Xpand auf die Checks-Ebene ausgelagert wurde. Dass ein korrekter Code nach Beendigung des Generierungsprozesses aus dem zugehörigen Modell entsteht, ist die Aufgabe des davor erfolgten Validierungsschrittes. Somit findet einerseits eine scharfe Teilung von Belangen statt, anderseits wird die Struktur der Templates einfach gehalten, was deren Bearbeitung (z.B. Parsen) erheblich vereinfacht. Dies führt zu einer deutlichen Optimierung der Leistungsfähigkeit der Codegenerierung.

An dieser Stelle ist die Behandlung der stufenweisen Einbettung der verschiedenen Validierungsrandbedingungen entlang der Ebenen der Werkzeugkette in Erinnerung zu rufen (vgl. Abschnitt 2.3). Wie erörtert, dient die Validierung auf Codegenerierungsebene u.a. dazu, spezifische domänennahe Randbedingungen festzulegen, die zugleich den Codegenerierungsprozess optimieren. Dieser Aspekt spiegelt sich in der gerade erwähnten engen Zusammenarbeit zwischen Xpand und den Checks-Ausdrücken wider. Beispielsweise wird in Visual ScatterUnit der Code eines Prozesses aus dem dazugehörigen Aktivitätsdiagramm für Testzwecke generiert. Daher wird das Aktivitätsdiagramm traversiert und zwar beginnend vom Startzustand bis zum Endzustand, währenddessen der Code aus den Templates für die Zwischenzustände emitiert wird. Dies setzt jedoch voraus, dass keine Zyklen im Diagramm enthalten sind, denn in diesem Fall würde der Codegenerierungsprozess endlos weiterlaufen. Der gesamte Generierungsprozess kann deshalb nicht gestartet werden, wenn die entsprechenden Bedingungen verletzt werden. Diese Prüfung während der Codegenerierung aus den Templates zu bewerkstelligen, wäre sehr komplex und ineffizient. Anhand von einfachen Checks-Ausdrücken kann dies im vorherigen Schritt der Validierung auf einfache und effiziente Weise ausgeschlossen, und so eine korrekte und effiziente Codegenerierung gewährleistet werden. Abbildung 11 zeigt ein Beispiel eines Xpand-Templates sowie deren Struktur. [Hent06]

Analog zu Checks wird zuerst das jeweilige Metamodell mit einer entsprechenden IMPORT-Anweisung eingebunden. Eine Xpand-Anweisung beginnt immer mit dem Zeichen << und endet mit >>. Das für ein Modellkonstrukt emittierte Artefakt wird innerhalb des Blocks der Anweisung DEFINE gehalten. In dem Beispieltemplate in Abbildung 11 wird Java Code generiert und in einer Datei ausgeschrieben, die – nach der Konvention von Java – denselben Namen wie die Klasse hat. Dies wird durch die Anweisung FILE bewerkstelligt. Der Name wird dabei dynamisch bestimmt. Im Beispieltemplate wird der Name der Klasse durch das entsprechende Attribut im Modellkonstrukt bestimmt.

Des Weiteren ist es durch die EXPAND-Anweisung möglich, dass ein Template von einem anderen aufgerufen wird, was deren Verschachtelung während des Generierungsprozesses ermöglicht. Ferner können mit der Sprache *Xtend*[22] [WebXtend] formulierte Erweiterungen (Extensions) durch die EXTENSION-Anweisung in ein Template dynamisch eingebettet werden.[23] Erweiterungen sind ein weiteres nütz-

[22] Analog zu Checks und Xpand bringt oAW einen extra Editor für Xtend-Anweisungen mit Funktionen zum Vollständigen von Code und dem Hervorheben von Syntaxelementen.

[23] Mit Xtends formulierte Anweisungen können ebenfalls in Checks-Blöcke eingebettet werden.

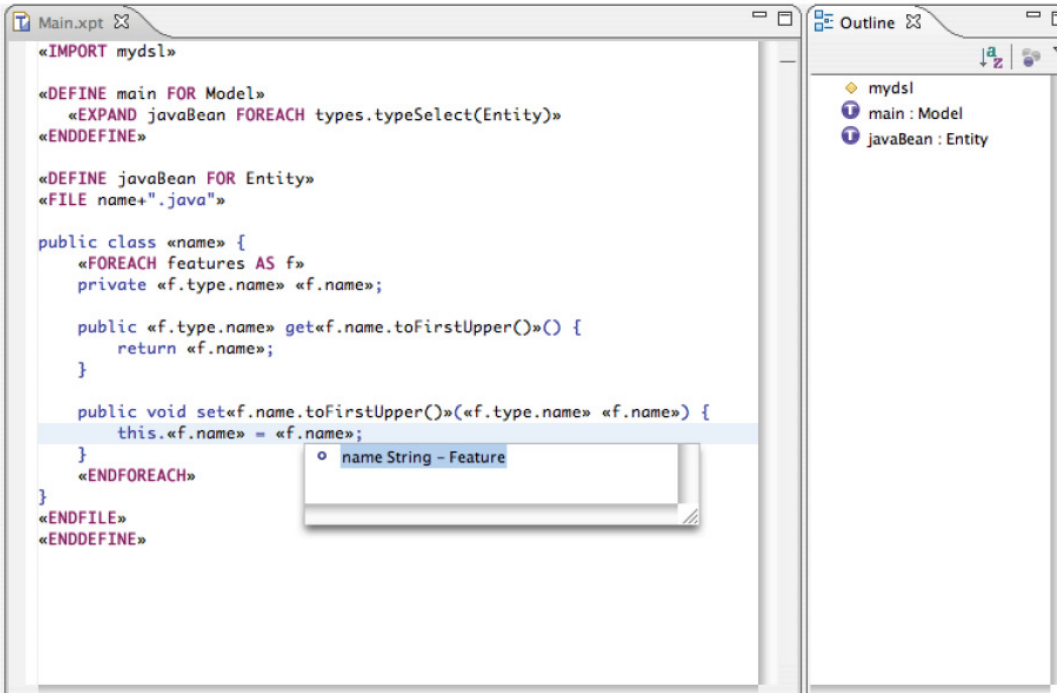


Abbildung 11 Die Template-Sprache Xpand von oAW [WeboAWDoc]

liches Feature, das von oAW unterstützt wird. Die Aufgabe von solchen Erweiterungen mittels Xtends ist, die Xpand-Sprache auf eine modulare Weise funktional bzw. operativ anzureichern – daher auch die Bezeichnung *Extensions*. Mit modular ist in diesem Zusammenhang gemeint, dass die durch Xtends erzielte Funktionalität separat und ohne einen Eingriff in den strukturellen Aufbau der Xpand-Sprache vonstatten geht. Die Xtends-Anweisungen bewirken, dass Operationen definiert werden, deren Resultate per Aufruf den Templates zur Verfügung gestellt werden. Die Definition der Operationen und die gesamte damit verbundene Logik bleiben den Templates verborgen. Ein Anwendungsbeispiel findet man in 4.3 bei der Behandlung der Codegenerierung in Visual ScatterUnit.

Vor dem Hintergrund der obigen Schilderungen wird ersichtlich, dass die technische Umsetzung der automatisierten Codegenerierung ein anspruchsvoller Prozess ist, der aus verschiedenen, aufeinander abgestimmten Schritten besteht, an deren Bewerkstelligung etliche Komponenten unterschiedlicher Werkzeuge beteiligt sind.

Um einen koordinierten Ablauf der verschiedenen Schritte zu gewährleisten, bietet oAW eine robuste und übersichtliche Lösung, die den Benutzer in die Lage versetzt, auf einfache Weise seine individuelle Konfiguration des Generierungsprozesses zu bestimmen. Hierzu liefert oAW standardmäßig eine XML-basierte Workflow-Engine. Dabei kann der Benutzer mit Hilfe von diversen Wizards eine zentrale XML-Datei erstellen, mit der die Reihenfolge der Arbeitsschritte des Generierungsprozesses sowie die daran beteiligten Werkzeuge festgelegt werden. Nun kann der Generierungsprozess vom Benutzer kontrolliert angestoßen werden, indem die Workflow-Engine die Datei sequenziell abarbeitet. In Abbildung 12 wird ein Beispiel einer solchen Workflow-Datei gezeigt. [Hent06]

```
<workflow>
    <property name="modelfile" value="model.scatterweb" />
    <property name="src-gen" value="src-gen" />

    <component class="org.openarchitectureware.emf.XmiReader">
        <metaModelPackage value="scatterweb.impl.ScatterwebPackageImpl" />
        <modelFile value="${modelfile}" />
        <outputSlot value="model" />
    </component>

    <component class="org.openarchitectureware.check.CheckComponent">
    <metaModel class="org.openarchitectureware.type.emf.EmfMetaModel">
    <metaModelPackage value="scatterweb.impl.ScatterwebPackageImpl" />
    </metaModel>
    <checkFile value="model::scatterweb::constraints::ScatterwebBatchErrors"/>
    <expression value="model.eAllContents"/>
    </component>

    <component id="generator" class="org.openarchitectureware.xpand2.Generator" skipOnErrors="true">
        <metaModel
            class="org.openarchitectureware.type.emf.EmfMetaModel">
            <metaModelFile value="metamodel/metamodel.ecore" />
        </metaModel>
        <expand value="templates::Root::Root FOR model" />

        <genPath value="${src-gen}" />
        <prSrcPaths value="src-gen"/>
        <prDefaultExcludes value="false"/>
    </component>

    <component class="org.openarchitectureware.xpand2.GeneratorAdvice" adviceTarget="generator">
        <advices value="templates::Logging"/>
    </component>
</workflow>
```

1 2 3 4

Abbildung 12 Die Workflow[24]-Konfiguration von ScatterFactory [Hent06]

1. Da der Ausgangspunkt des Generierungsprozesses das eingegebene Modell ist, wird dieses als erstes durch das Property-Element in der Workflow-Datei festgelegt. Zugleich wird auch das Zielverzeichnis angegeben, in das die aus dem zuvor festgelegten Modell generierten Softwareartefakte (das Ergebnis des Generierungsprozesses) geschrieben werden.
2. Um das im ersten Schritt angegebene Modell bearbeiten zu können, muss, wie bereits geschildert, das eingegebene Modell auf Basis des Metamodells

[24] Die Workflow-Datei hat immer die Endung .oaw.

instanziiert werden.[25] Durch *metaModelPackage* wird der Pfad des Metamodells mitgeteilt. Um nun das instanziierte Modell für die übrigen Komponenten von oAW während des Generierungsprozesses handhabbarer zu machen, wird es mit Hilfe von *outputSlot* durch eine Referenz namens *model* repräsentiert, über die dann auf das instanziierte Modell zugegriffen werden kann.

3. Bevor die Codegenerierung auf Basis der definierten Templates angestoßen wird, muss die Modellintegrität gewährleistet sein. Dafür müssen die mit Checks spezifizierten Randbedingungen in Betracht gezogen werden. Hierzu muss der Zielort der Datei, in der die Checks-Bedingungen enthalten sind, angegeben werden. Nun müssen die Checks-Bedingungen auf Basis der Modellkonstrukte validiert werden, d.h. jedes Objekt im instanziierten Modell muss bezüglich seiner Checks-Bedingungen geprüft werden. Hier kommt die in Schritt 1 definierte Modell-Referenz *model* zum Zuge. Denn durch *model.eAllContents* werden sämtliche Objekte des instanziierten Modells aufbereitet und zur Verfügung gestellt.
4. Nun kann der Code der einzelnen Objekte des instanziierten Modells aus den korrespondierenden Templates emitiert werden. Zu diesem Zweck muss das Wurzeltemplate angegeben werden, mit dem der Generierungsprozess startet. Hierzu dient das expand Attribut. Um Add-ons neben den Software Artefakten zu generieren, kann die GeneratorAdvice oAW-Komponente aktiviert werden, die zur Aufgabe hat, gesonderte Templates (z.B. modellunabhängige Templates) in die Modell-zu-Code-Transformierung dynamisch zu binden.

Dadurch können beispielsweise Readme-Dateien mit Hilfe des Logging.xpt Template für den Nutzer mitgeneriert werden. Die in der Workflow-Datei auftretenden Elemente spiegeln die zahlreichen beteiligten Werkzeuge und Technologien wider, die in diesem Kapitel behandelt wurden und aus denen die ScatterClipse Werkzeugkette aufgebaut ist. Die folgende Abbildung 13 entspricht der Abbildung 2 in Abschnitt 2.1, allerdings werden hier statt der verwendeten Konzepte im MDSD-Prozess die zur Realisierung dieser Konzepte eingesetzten Rahmenwerke und Technologien dargestellt.

Um nun die Tauglichkeit dieser Rahmenwerke für den MDSD-Prozess näher zu betrachten, ist es an dieser Stelle unerlässlich, einen tieferen Einblick in den Einsatz der Rahmenwerke zu geben, der eine aussagekräftige Beurteilung ermöglicht. Dies soll anhand des Erfahrungsberichts über ein selbst entwickeltes Rahmenwerk im folgenden Kapitel geleistet werden. Dieser dient dem Zweck, die Tauglichkeit der behandelten Rahmenwerke im Kontext des MDSD-Prozesses zu demonstrieren und ihre Robustheit zu beurteilen. Bei dem folgenden Erfahrungsbericht handelt es sich jedoch nicht um eine Beurteilung der ScatterClipse-Plattform. Eine fundierte Evaluation von ScatterClipse findet sich in Kapitel 8.

[25] Im Falle von SactterFactory handelt es sich wie erwähnt um ein mit EMF definiertes Ecore-Metamodell. Daher wird *emf.XmiReader* in oAW aktiviert.

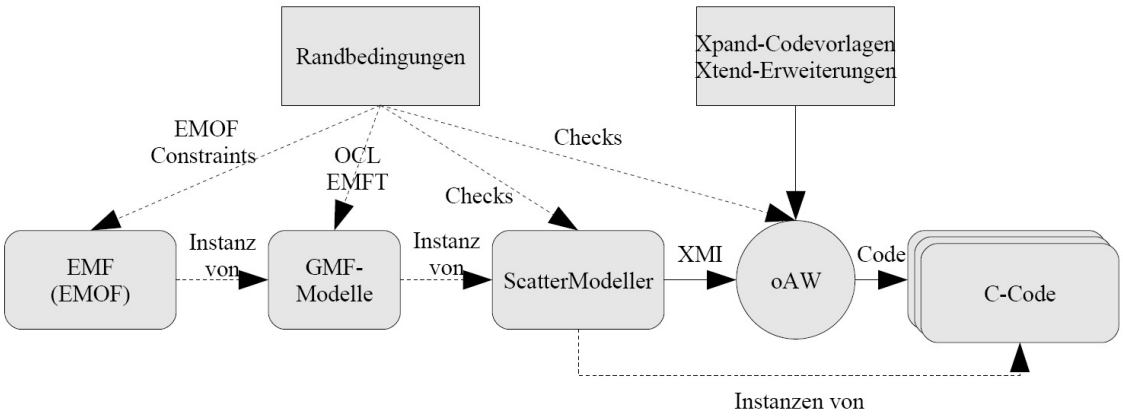


Abbildung 13 Die in ScatterClipse eingesetzten Werkzeuge und Technologien im Kontext des MDD-Prozesses

2.6. Zusammenfassung

Das Kapitel widmete sich dem modellgetriebenen Entwicklungsprozess sowie der Werkzeuge, die zur Realisierung des Prozesses im Rahmen der ScatterClipse Werkzeugkette eingesetzt werden. Zunächst wurde auf den in der Arbeit verfolgten architekturzentrierten Ansatz eingegangen und dieser im modellgetriebenen Paradigma eingeordnet. Ferner wurde dieser Ansatz der modellgetriebenen Architektur gegenübergestellt, um zu zeigen, warum sich gerade die Architekturzentrierung für die Herausforderung der Automatisierung als Lösung optimal eignet. Ferner wurden die zur Realisierung der architekturzentrierten generativen Infrastruktur eingesetzten Technologien und Werkzeuge, von denen jedes für eine Phase des modellgetriebenen Prozesses zuständig ist, genauer unter die Lupe genommen. Mit der Behandlung jedes Werkzeuges wurden zugleich die in der jeweiligen Phase stattfindenden Abläufe und Aktivitäten geschildert.

Das Resultat eines Werkzeuges bildet oft die Eingabe des darauf folgenden. Auf Basis des mit dem EMF erstellten Metamodells wurde gezeigt, wie ein generativer domänenspezifischer grafischer Editor mit Hilfe von GMF automatisiert erzeugt werden kann. Da die domänenspezifischen Sprachen (DSL) nicht in einem Guss entworfen, sondern eher iterativ konstruiert werden, zeigt sich die Stärke der harmonischen Integration beider Werkzeuge. So ist es möglich, den generativen DSL-Editor (GMF) nach jeder Verfeinerung des Metamodells (EMF) automatisiert zu generieren.

Die durch den Editor modellierten Konstrukte der Domänensprache werden vom Generatorrahmenwerk openArchitectureWare (oAW) eingelesen, validiert und im Anschluss auf deren korrespondierende Templates abgebildet, um dann den Codegenerierungsprozess anzustoßen. Bei der Darstellung dieses Prozesses standen der Templates-gestützte Codegenerierungsprozess und dessen Bewerkstelligung mit Hilfe der von oAW gelieferten unterschiedlichen Arten von Templates im Mittelpunkt. Des Weiteren wurde auf die Validierung der Modelle der Domänensprache eingegangen, die durch die unterstützte Integration zwischen EMF/GMF auf der einen Seite und dem oAW Generatorrahmenwerk auf der anderen ermöglicht wird. Diese Integration verdankt sich dem vom oAW mitgelieferten GMF-Adapter. Dabei

wurde geschildert, wie durch einen Watchdog alle Änderungen der Modellkonstrukte im Editor zur Modellierungszeit überwacht werden und gleichzeitig in Abhängigkeit der vollzogenen Modelländerungen die Erfüllung von zentral gehaltenen Randbedingungen überprüft wird. Ferner wurde illustriert, wie solche Randbedingungen mit Hilfe der ebenfalls von oAW mitgelieferten Sprache Checks formuliert werden können. Die für die Konstruktion der Werkzeugkette verwendeten Rahmenwerke sind ausschließlich Eclipse-Projekte. Dies ermöglicht eine optimale Integration der Werkzeuge, wodurch eine einwandfreie Zusammenarbeit gemäß dem modellgetriebenen Paradigma erzielt wird. Alle verwendeten Rahmenwerke sind Open Source-Projekte.

3. Der Pfad des Anwendungstests: Visual ScatterUnit

Angenommen, es soll ein System zur Steuerung einer sich selbst konfigurierenden HiFi-Anlage getestet werden, wobei das System die gesamten Räumlichkeiten des Hauses abdecken soll. Hierzu wird jeder Raum des Hauses mit einem Radioempfänger sowie einem Lautsprecher ausgestattet. Die Hauptaufgabe des HiFi-Steuerungssystems ist es, zu gewährleisten, dass genau diejenigen Komponenten des Systems eingeschaltet sind, die sich in dem Raum befinden, den eine Person gerade betritt bzw. in dem sie sich momentan aufhält. Dazu muss im zuletzt betretenen Raum derjenige Sender automatisch ermittelt werden, den die Person im zuvor verlassenen Raum gehört hat. Ein solches System besteht aus einer Vielzahl von an verschiedenen Stellen platzierten drahtlosen Sensorknoten, wobei die Sensorknoten eines Raumes mit dem Radioempfänger des benachbarten verbunden sind. Die Sensorknoten eines Raumes können somit den gerade von der Person gehörten Sender im benachbarten Raum ermitteln, dessen Radioempfänger ausschalten und den eigenen beim Zeitpunkt des Betretens des eigenen Raumes anschalten. Um den Zeitpunkt des Verlassens des benachbarten und des Betretens des eigenen Raumes zu ermitteln, müssen bestimmte Knoten mit Bewegungssensoren ausgestattet sein. Solche Bewegungssensoren können ihrerseits die betreffenden Knoten über ein solches Ereignis benachrichtigen.

Eine einfache Möglichkeit ein solches HiFi-System zu testen, wäre schlicht, durch die verschiedenen Räume des Hauses zu wandern und währenddessen die Radiostationen ständig zu wechseln, um zu sehen, ob alles wie erwünscht funktioniert. Das Testen eines solchen Systems in der Ära des *Ubiquitous and Pervasive Computing* (U & P Computing)[26] [Weis91], z.B. in einem *Smart Home,*[27] muss jedoch automatisiert vonstatten gehen.

Um die angestrebte Testautomatisierung zu ermöglichen, müssen bestimmte Aktionen des Testverlaufs – die Bewegung durch das Haus – simuliert werden. So ist bei diesem Testszenario die Simulation der zu erfassenden Daten des Bewegungssensors und der Aktion zum Umschalten der Radiostationen vonnöten. Ferner muss das Voranschreiten des auf den verschiedenen Sensorknoten verteilten Testverlaufs koordiniert und beobachtet werden. Solche Mechanismen zum Zweck der Testautomatisierung benötigen ein entsprechendes Rahmenwerk.

Der Pfad des automatisierten Anwendungstests in ScatterClipse ist durch das generative modellgetriebene Rahmenwerk Visual ScatterUnit realisiert. [Kame07, Kame08]. Das Rahmenwerk stellt ein umfassendes Verfahren zur Verfügung, das Komponenten- und Systemtests der auf den Sensorknoten laufenden Anwendungen

---

[26] „Ubiquitous computing (ubicomp) is a post-desktop model of human-computer interaction in which information processing has been thoroughly integrated into everyday objects and activities. As opposed to the desktop paradigm, in which a single user consciously engages a single device for a specialized purpose, someone "using" ubiquitous computing engages many computational devices and systems simultaneously, in the course of ordinary activities, and may not necessarily even be aware that they are doing so.“ [WebPerv]

[27] „Home automation (also called domotics) is a field within building automation, specializing in the specific automation requirements of private homes and in the application of automation techniques for the comfort and security of its residents. Although many techniques used in building automation [...] are also used in home automation, additional functions in home automation can include the control of multi-media home entertainment systems, automatic plant watering and pet feeding, automatic scenes for dinners and parties, and a more user-friendly control interface.“ [WebHome]

automatisiert ermöglicht. Um nun den oben geschilderten Testverlauf zu steuern, ist es unerlässlich, die verteilte Ausführung des Testfalls[28] (test case) zu koordinieren. Dafür muss ein entsprechender Synchronisationscode geschrieben werden, was ein schwieriges Unterfangen ist, da die manuelle Erstellung eines solchen Codes oft komplex und daher zeitaufwendig ist.

Wird nun diese Aufgabe automatisiert erledigt und dem Entwickler durch einen Codegenerator abgenommen, trägt dies erheblich zur Steigerung der Effizienz des Testprozesses bei. Dies ist die Schlüsselidee des im Rahmenwerk verfolgten modellgetriebenen Ansatzes. Daher werden Testszenarios durch ihre korrespondierenden formalen Modelle repräsentiert und anschließend nach deren Validierung in den Codegenerator eingegeben, um den Code des Testfalls zu generieren und auf den Sensorknoten auszubringen. Der Anwendungstester bzw. Entwickler muss sich lediglich auf den Entwurf des Testszenarios konzentrieren und ist daher weniger mit technischen Belangen beschäftigt.

Nach der Ausführung des Testfalls werden die Testergebnisse gesammelt, um anschließend in den zuvor erstellten Modellen visualisiert zu werden. Die formalen Modelle vereinfachen ebenso die Aufgabe der Lokalisierung entstehender Fehler in der zu testenden Anwendung. An dieser Stelle kommt die zweite Schlüsselidee zum Tragen: der visuelle kognitive Debuggingprozess, der iterativ vom Entwickler in Abhängigkeit vom Testverlauf betrieben werden kann. Bei jeder Iteration des Testverlaufs werden die Modelle verfeinert und anschließend zur Automatisierung in den Codegenerator eingegeben. Nach Beendigung jeder Iteration bzw. Testfallausführung werden genauere Informationen zum Verhalten der Anwendung gesammelt und anschließend in den verfeinerten Modellen visualisiert. Demzufolge können aussagenmächtigere Hypothesen vom Softwaretester hinsichtlich der Versagensursache im Code der Anwendung getroffen werden. Der zweigleisige Einsatz der Modelle – bestehend aus der Kombination der Visualisierung einerseits und dem Zweck der Automatisierung andererseits – stellt die Innovation des Test-Rahmenwerks dar.

An dieser Stelle stellt sich die Frage, wie sich das eben Gesagte mit dem eingangs erwähnten Szenario der Navigation durch verschiedene Räume vereinbaren lässt. Mit anderen Worten, wie sieht der Ablauf eines möglichen Testszenarios beim Einsatz von Visual ScatterUnit aus?

Jedes Testszenario ist durch einen entsprechenden Testfall zu realisieren. Somit geht der Realisierung des Testfalls stets der Entwurf eines Testszenarios voran, das vom Testfall im Kontext der Testinfrastruktur bewerkstelligt werden soll. Im Entwurf des Testszenarios müssen daher die Aktionen und Ereignisse zur Durchführung des Tests zum Ausdruck gebracht werden. Des Weiteren muss durch das Testszenario vorgegeben werden, auf welchem Sensorknoten und zu welchem Zeitpunkt eine Aktion während des Tests ausgeführt werden muss. Mit anderen Worten, die Informationen zur Reihenfolge der verteilten Ausführung der Aktionen sowie deren Zuordnung zu den jeweiligen Knoten müssen aus dem Testszenario hervorgehen, damit auf dieser Basis der zugehörige Testfall konkretisiert werden kann.

Um den Testfall zu implementieren, sind für die Testschicht des Rahmenwerks ScatterUnit zwei Module, genannt *node scripts*, vonnöten, die auf den jeweiligen Sensorknoten ausgeführt werden. Die Skripte werden aus dem formalen Modell des

[28] Siehe Glossar.

zugehörigen Testfalls mit Hilfe des Codegenerators automatisiert generiert. Jedes dieser Skripte simuliert die von einem Sensorknoten in Abhängigkeit von seiner Rolle erfassten Daten. Demnach simuliert das auf dem ersten Sensorknoten ausgeführte Skript die Daten der folgenden Aktion: Die Person betritt den (ersten) Raum und schaltet die Anlage an. Das auf dem zweiten Knoten laufende Skript simuliert die Aktion, dass die Person den ersten Raum verlässt und den zweiten betritt. Um die Reihenfolge des Auftretens der voneinander abhängigen Aktionen zu gewährleisten, muss der Ablauf der Aktionen nach ihrem Eintreten koordiniert werden. Die Erstellung des dazu benötigten Codes wird nun gemäß dem modellgetriebenen Paradigma an den Codegenerator des Visual ScatterUnit-Rahmenwerks delegiert.

Der Code wird aus dem formalen Modell des Testfalls generiert, der den Verlauf des Testszenarios repräsentiert. Abschließend werden die von den beiden Sensorknoten mit Hilfe der Skripte während des Testverlaufs gesammelten Daten der oben beschriebenen Aktionen vom Rahmenwerk analysiert und in den Modellen visualisiert. Beim Testverlauf werden auch die Fehler protokolliert. Es stellt sich die Frage, wie man die betreffenden Codestellen, die die Fehler verursacht haben, in der Anwendung lokalisieren kann. Zu diesem Zweck werden die Modelle iterativ verfeinert, um graduell die für die Fehler verantwortlichen Codestellen zu isolieren. Bei jedem solchen Schritt wird der notwendige Infrastruktur-Code an den Codegenerator erneut delegiert und die gegebenenfalls neu entstehenden Fehler erneut in den dazugehörigen Modellkonstrukten visualisiert. Wie dies im Rahmen von Visual ScatterUnit genau vonstatten geht, wird im Laufe des Kapitels ausführlich dargestellt.

3.1. Architektur des Rahmenwerks Visual ScatterUnit

Zu Beginn wurde die Testinfrastruktur ScatterUnit als erster Baustein der Architektur des Rahmenwerks entwickelt. ScatterUnit ermöglicht eine effiziente Testautomatisierung der auf den Sensorknoten laufenden Anwendungen, was deren System- und Integrationstests umfasst. Dabei kann der Testverlauf sich auf eine Vielzahl von beteiligten Sensorknoten erstrecken. Wie ein Debugger kann ScatterUnit Testszenarien ablaufen lassen, in die viele Sensorknoten involviert sind. Diese Testinfrastruktur ist vergleichbar mit der JUnit für Java Anwendungen, die es ermöglicht, den Testverlauf zu kontrollieren und dem Ablauf des Tests zu folgen, um zu entscheiden, ob eine Anwendung versagt hat oder nicht.

Um den besonderen Anforderungen der Domäne drahtloser Sensornetze gerecht zu werden, müssen bestimmte Entwurfsentscheidungen auf der Architekturebene getroffen werden. Man denke etwa an die schwachen Ressourcen oder den höheren Parallelitätsgrad der Anwendungen in einer solchen Domäne. Daher stellt ScatterUnit mehrere Dienste zur Verfügung, die für den Testzweck vielfältiger WSN-Anwendungen oft mehrfach in Anspruch genommen werden, beispielsweise für die Erfassung der gesammelten Sensordaten und deren Protokollierung oder die Ermittlung der Topologie des Netzes, um den Aspekt der Mobilität zu unterstützen. Darüber hinaus werden vielfältige Dienste unterstützt, die in 4.4 ausführlicher behandelt werden. Wichtig ist an dieser Stelle, dass durch die von ScatterUnit angebotenen Dienste der extensive Test der auf den Sensorknoten laufenden Anwendungen unbeeinträchtigt bleibt.

Obwohl ScatterUnit die angestrebte Automatisierung ermöglicht, bleibt die Implementierung eines Testfalls eine für den Anwender zeitaufwendige und komplexe

Tätigkeit. Dies liegt vor allem an der vorhin erwähnten Eigenschaft der WSN-Domäne, nämlich dem höheren Verteilungsgrad der Anwendungen. Denn nur in diesem Kontext können Testfälle für die auf den Sensorknoten laufenden Anwendungen erstellt werden.

Hierfür muss die Ausführung des Testfalls koordiniert werden, was einen zu diesem Zweck eigens zu implementierenden Code benötigt. Um die Arbeit für den Anwender deutlich zu erleichtern, wird diese Aufgabe, wie bereits erwähnt, in den Codegenerator verlagert. Zu diesem Zweck wurde ScatterUnit[29] mit der modellgetriebenen Visual ScatterUnit ergänzt, wodurch die modellgetriebene Ausführung der Testfälle ermöglicht wird. Des Weiteren reduziert die Modellierung die inhärente Komplexität der Entwicklung der Testfälle, die durch die Steuerung der Verteilung sowie die begleitende Koordination der im Test ablaufenden Aktionen verursacht wird. Infolgedessen erhöht sich die Qualität des Testprozesses erheblich, da sich der vom Entwickler manuell implementierte Code massiv reduziert. Dies erhöht die Robustheit und Effizienz des Testens entsprechend. Abbildung 14 zeigt die für die Realisierung des automatisierten Testprozesses notwendige abgestimmte Zusammenarbeit zwischen den zwei Ebenen.

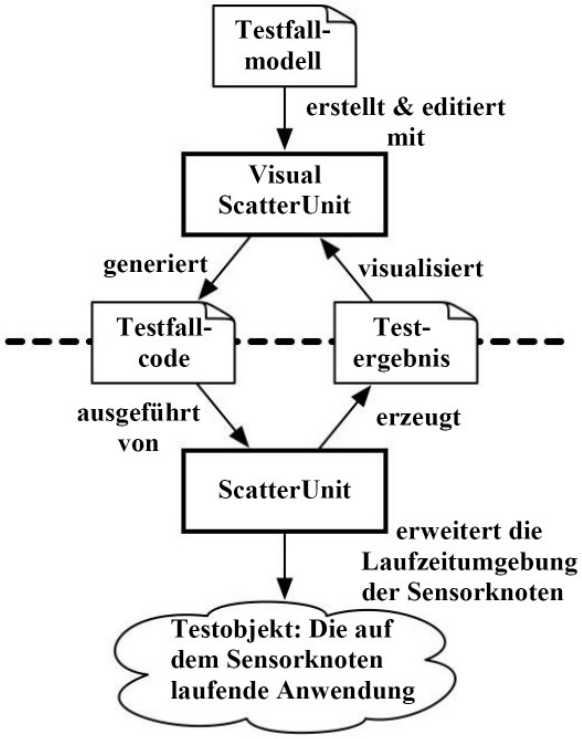


Abbildung 14 Zusammenarbeit zwischen ScatterUnit und Visual ScatterUnit

[29] Die Bezeichnung ist von JUnit inspiriert. „JUnit is a unit testing for the Java programming language. Created by Kent Beck and Erich Gamma, JUnit is one of the xUnit family of frameworks that originated with Kent Beck's SUnit. JUnit has spawned its own ecosystem of JUnit extensions." [Web-JUnit]

Der Testprozess besteht aus folgenden Aktivitäten:

- Wahl des passenden Testszenarios für die zu testende Anwendung
- Erstellung des entsprechenden formalen Modells des Testfalls, durch den das entworfene Testszenario realisiert wird
- Vervollständigung der Entwicklung des Testfalls. In diesem Schritt wird Code manuell implementiert.
- Ausführung des Tests auf den Sensorknoten
- Analyse und Evaluierung der Testergebnisse
- Visualisierung der Testergebnisse im Testfallmodell
- Verfeinerung des Testfallmodells, um die Fehler zu isolieren bzw. zu lokalisieren. Hierfür werden auf Basis der visualisierten Testergebnisse Hypothesen bezüglich der Fehlerursache aufgestellt.
- iteratives Vorgehen bis alle Fehler beseitigt sind

Der Entwickler kann während der Analyse der Testergebnisse zwei mögliche Intentionen haben. Einerseits liegt sein Fokus generell auf der Feststellung von Fehlern, andererseits sind Erkenntnisse von Interesse, die die Entstehung von Fehlern nachvollziehbar machen. Solche Informationen kann der Entwickler verwenden, um die Ursachen von Fehlern zu lokalisieren. Beide Fähigkeiten, die Fehlererfassung und deren Lokalisierung, müssen unterstützt werden und eng zusammen arbeiten.

Im Rahmen der Testfallerstellung ist vor allem die Verringerung der dazu benötigten Entwicklungszeit entscheidend, da die Gewährleistung der Qualität der zu testenden Anwendung durchaus die Implementierung mehrerer Testfälle verlangen kann. Was die Lokalisierung der Fehler anbelangt, steht vor allem die Reduktion der dabei entstehenden Komplexität im Fokus. Da die Fehlerlokalisierung präzisere Informationen verlangt, müssen hierzu die betreffenden Testfälle graduell erweitert werden, um nähere Informationen zur Fehlerlokalisierung zu gewinnen. Die Komplexität muss also reduziert werden, um den Entwickler in die Lage zu versetzen, die Testfälle entsprechend flexibler zu modifizieren. Diese Komplexitätsreduzierung wird durch Automatisierung erreicht. Welche Abläufe und Aktivitäten, durch die erzielte Automatisierung beim Einsatz von Visual ScatterUnit stattfinden und wie die einzelnen Komponenten des Rahmenwerks zusammenarbeiten und untereinander ihre Daten austauschen, zeigt die Abbildung 15.

Zu Beginn modelliert der Entwickler den Ablauf des Tests, wodurch der Fluss und die Gliederung der für den Testverlauf erforderlichen Aktivitäten festgelegt werden. Die Modellierung geschieht mit Hilfe der speziell zu diesem Zweck entworfenen Domänensprache (DSL). Dazu steht ein GMF-basierter grafischer Editor zur Verfügung. Abbildung 16 zeigt einen Bildschirmabzug des Editors. Die Konstruktion der Domänensprache (d.h. die Festlegung der Syntax und der Semantik) basiert auf einem Ecore Meta-Modell, das gemäß dem EMF Meta-Meta-Modell definiert wurde. Die Modellierung des Testverlaufs in der Domänensprache orientiert sich stark an der Notation der Aktivitätsdiagramme in der UML. [Rupp07, S. 235-365] Aktivitätsdiagramme eignen sich gut für diesen Zweck, da es sich hauptsächlich um die Steuerung des Testverlaufs handelt.

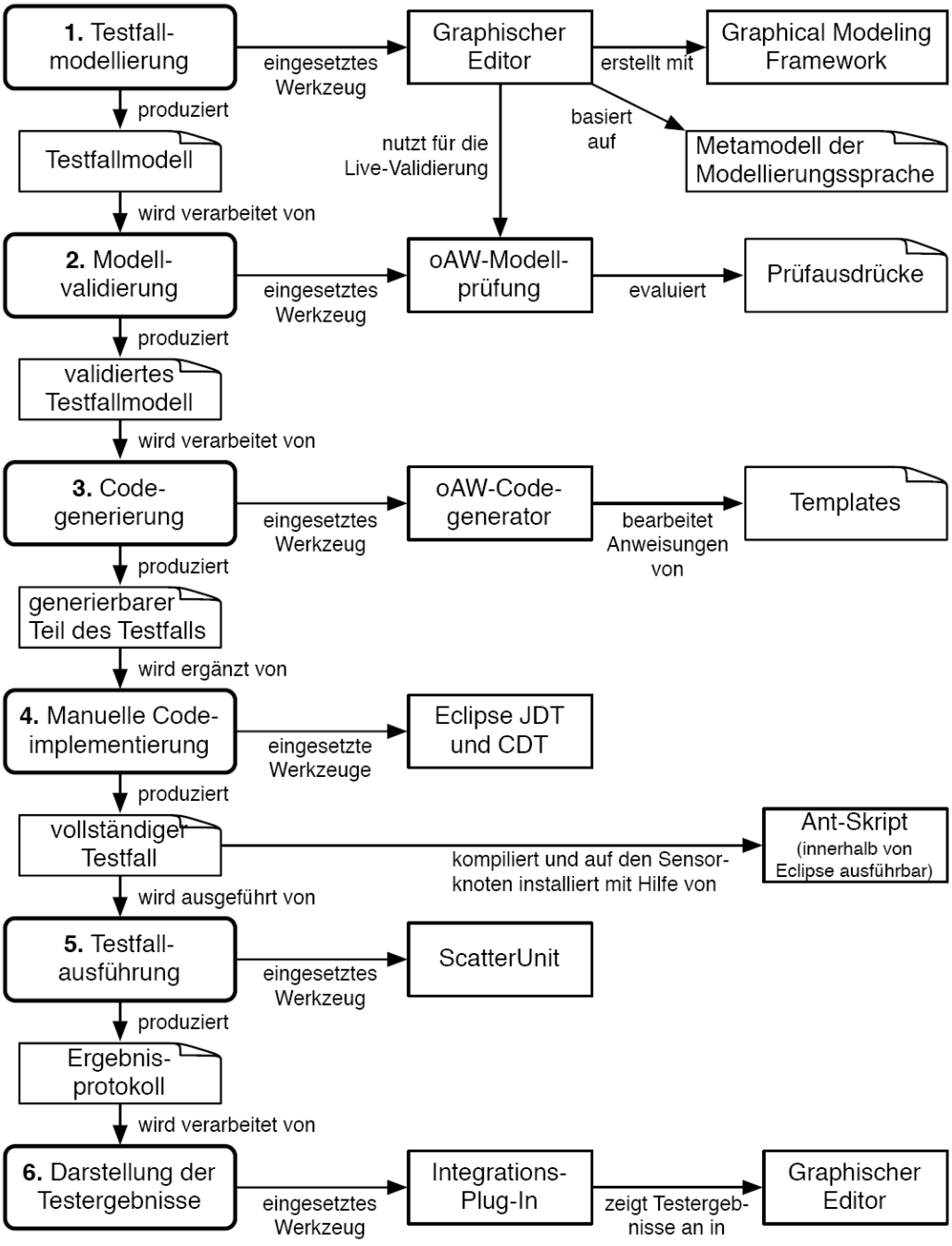


Abbildung 15 Aktivitäten beim Einsatz von Visual ScatterUnit [Kame08]

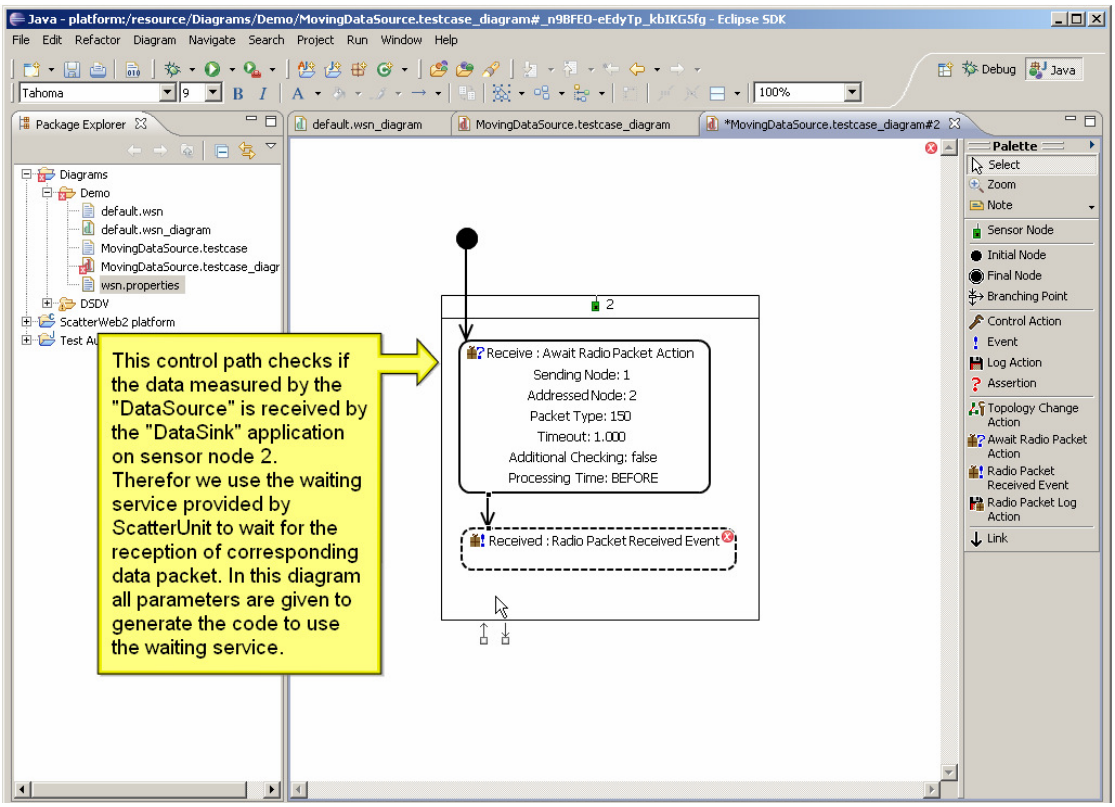


Abbildung 16 Der generative Editor von Visual ScatterUnit

Der Einsatz von eng an die UML angelehnten Aktivitätsdiagrammen innerhalb der eigenen DSL (statt einer eins zu eins Übernahme der Diagramme aus der UML selbst bzw. aus deren Werkzeugen) ermöglicht einerseits einen geläufigen Modellierungsstandard, da man sich an die Notation als Standard hält, andererseits wird eine maßgeschneiderte Verwendung dieser Modellinstanzen speziell für die eigene Modellierungsbelange im Kontext der selbstentworfenen Domänensprache erreicht. Demgemäß ist es beispielsweise möglich, interaktiv und flexibel mit einfachem Doppelklick die Aktivitätsdiagramme zu verfeinern und innerhalb ihrer Aktionen zu navigieren, um die Fehler zu isolieren, die Testergebnisse zu visualisieren und an richtiger Stelle in die Modelle einzubetten. All dies wäre werkzeugbedingt mit herkömmlichen UML-Diagrammen kaum machbar. Hier zeigt sich zum einen die Stärke des Ansatzes, generative Editoren auf Basis des DSL-Meta-Modells einzusetzen und zum anderen *domänenspezifisch* zu modellieren. Dies zeigt sich ebenfalls anhand der domänenspezifischen Just-in-time-Validierung (auch Live-Validierung genannt) der Modelle, die durch die Definition mit Hilfe der Checks-Sprache ermöglicht wird. Dies wird, wie bereits in 2.5 beschrieben, durch den von openArchitectureWare gelieferten GMF-Adapter bewerkstelligt. Abbildung 17 zeigt ein Beispiel eines Aktivitätsdiagramms zur Modellierung eines Testverlaufs mit einer Meldung der Live-Validierung im grafischen Editor sowie die entsprechende detaillierte Meldung in der Eclipse *Problem View* im unteren Bereich des Editors.

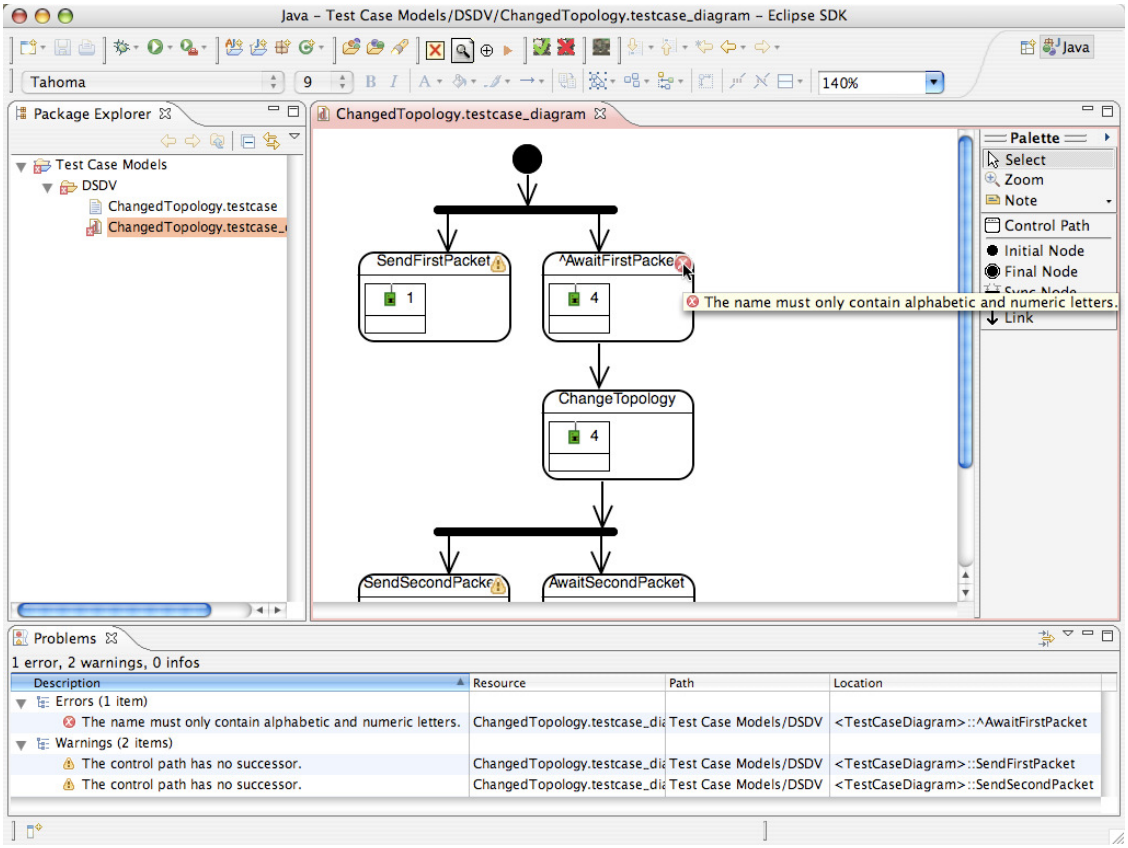


Abbildung 17 Modellierung des Testverlaufs mit Hilfe von Aktivitätsdiagrammen [Kame08]

Nach der Modellvalidierung kann nun der Codegenerierungsprozess angestoßen werden. Allerdings bleibt der generierte Code des Testfalls immer noch unvollständig, da bestimmte anwendungsspezifische Codesegmente überwiegend manuell vom Entwickler implementiert werden müssen. Da bestimmte Codeartefakte direkt auf den Sensorknoten ausgeführt werden, sind diese in der Programmiersprache C zu implementieren. Anders verhält es sich, was Visual ScatterUnit anbelangt.

Weil das Rahmenwerk Eclipse-basiert ist, erfolgt seine Implementierung vollständig in der Programmiersprache Java. Um nun den Datenaustausch zwischen den C-basierten ScatterWeb-Sensorknoten und der Java-basierten Visual ScatterUnit zu ermöglichen, wird die für eingebettete Systeme entwickelte COM-PORT [WebJavaCom] Bibliothek[30] verwendet. Dadurch wird die USB-Schnittstelle des Rechners, auf dem das Rahmenwerk läuft und an dem somit die Sinke (z.B. eGate)

[30] "The Java Communications 3.0 API is a Java extension that facilitates developing platform-independent communications applications for technologies such as Smart Cards, embedded systems, and point-of-sale devices, financial services devices, fax, modems, display terminals, and robotic equipment.
The Java Communications API (also known as javax.comm) provides applications access to RS-232 hardware (serial ports) and limited access to IEEE-1284 (parallel ports), SPP mode." [WebJavaComm]

angeschlossen ist, als Java-Objekt repräsentiert, über das die Daten textbasiert ausgetauscht werden.[31]

Sobald der Testfall vollständig erstellt wurde, kann er mit Hilfe der Eclipse-basierten Umgebung C-Development-Tool (CDT) kompiliert werden. Anschließend wird mit Hilfe des Ant-Plugins das als Hex-File vorliegende *Code-Image* auf die jeweiligen Sensorknoten entweder seriell oder drahtlos ausgebracht.[32]

Im Weiteren wird nun der Test auf den Knoten ausgeführt und währenddessen werden die für die Testevaluierung relevanten Informationen protokolliert. Nach der Beendigung des Tests werden die soeben protokollierten Informationen an das Rahmenwerk mittels der COM-PORT Bibliothek übermittelt. Die Daten werden an die entsprechende Überprüfungsroutine übergeben, die sie ihrerseits interpretiert und aufbereitet. Dabei markiert die Prüfroutine die gesammelten Daten des Testprotokolls entsprechend der aufgefundenen Fehler. Nun können die analysierten und markierten Daten des Testprotokolls vom Visualisierungs-Plugin geladen werden, wodurch die Testdaten an den richtigen Stellen innerhalb der formalen Modelle des eben protokollierten Testfalls im grafischen Editor visualisiert werden.

3.2. Die Testautomatisierungsschicht ScatterUnit

Für den Entwurf der Testautomatisierungsschicht ScatterUnit spielen die Anforderungen der Einsatzgegebenheiten an das Rahmenwerk in Bezug auf die ScatterWeb Sensorknoten eine entscheidende Rolle. Hierbei ist das Spektrum der mit dem Rahmenwerk potentiell zu testenden Anwendungen von zentraler Bedeutung. [Akyi02] Eine typische WSN-Anwendung (wie es auch in ScatterWeb der Fall ist) besteht hauptsächlich in der Erfassung der Umgebungsdaten über eine bestimmte Zeitperiode und anschließend der Evaluierung der gesammelten Daten auf dem Rechner bzw. einem designierten Knoten (vgl. Abbildung 18).

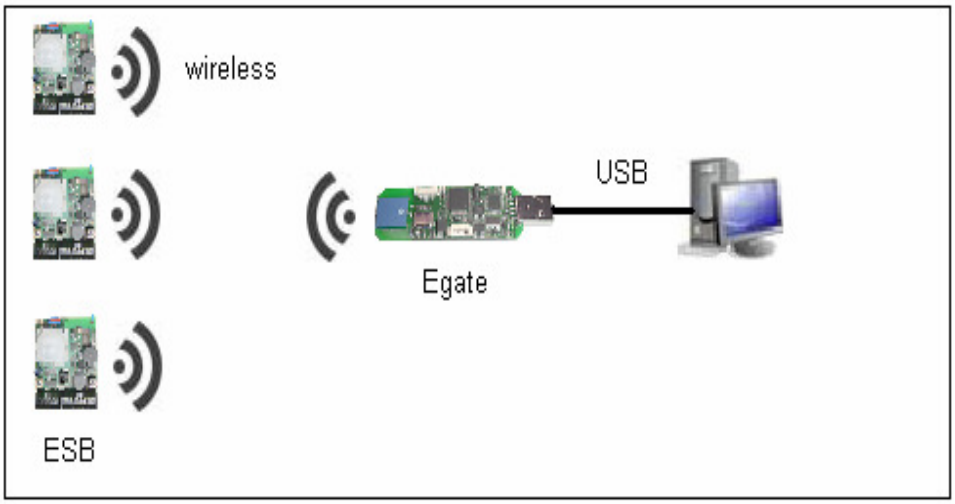


Abbildung 18 Die beteiligten Komponenten einer typischen ScatterWeb-Anwendung [Ding07]

[31] Mehr zur technischen Realisierung in 6.1 bei der Behandlung der Bibliothek ScatterWeb for Java.

[32] Zu diesem Zweck werden die entsprechenden Komponenten des WSN-VerwaltungsPfades von ScatterClipse eingesetzt. Mehr dazu in Kapitel 7. An dieser Stelle zeigt sich noch einmal die ergänzende Zusammenarbeit der verschiedenen Komponenten der ScatterClipse Werkzeugkette.

Um einen Testfall für das obige Szenario zu realisieren, muss die Testinfrastruktur in der Lage sein, die von den Sensorknoten während der Ausführung der zu testenden Anwendung übermittelten Daten zu analysieren, um feststellen zu können, ob der Test fehlerfrei verlaufen ist oder nicht. Dadurch lassen sich Rückschlüsse auf die Korrektheit der Funktionalität der Anwendung gewinnen. Daher muss die Testinfrastruktur die zu testende Anwendung mit Mechanismen erweitern, die im Allgemeinen den Testfall in die Lage versetzen, die während der Anwendungsausführung ablaufenden Aktionen bzw. auftretenden Ereignisse zu protokollieren, um auf dieser Basis das Verhalten der Anwendung nachvollziehen zu können.

Für die Integration solcher Mechanismen in die Laufzeitumgebung der Sensorknoten müssen die entsprechenden Ressourcen (vor allem Rechenzeit) auf Sensorarchitekturebene in Anspruch genommen werden. Dies darf jedoch das initiale Verhalten der zu testenden Anwendung in keiner Weise beeinträchtigen bzw. manipulieren. Unerwünschte Folgen dieses Eingriffs müssen daher in Betracht gezogen werden. [Cunh01] Dies ist eine essentielle Entwurfsanforderung, da es sogar dazu kommen kann, dass durch einen solchen Effekt die Ausführung der Anwendung fehlschlägt.

Dies ist insbesondere der Fall, wenn die Anwendung zeitkritisch (z.B. bei sehr geringer Reaktionszeit auf auftretende Ereignisse) ausgeführt werden muss. Des Weiteren soll keine Beeinträchtigung verursacht werden, wenn ein Dienst getestet wird, der auf verschiedenen Sensorknoten gleichzeitig läuft. Dies ist z.B. der Fall, wenn es sich beim Testobjekt um ein Routingprotokoll handelt, was ein typisches und weit verbreitetes Szenario im Kontext drahtloser Sensornetze ist.

Folglich wird ein dezentralisierter Steuerungsmechanismus zur Ausführung des Testverlaufs in der Testschicht ScatterUnit verfolgt, [Ulri99] um diesen Anforderungen gerecht zu werden und unerwünschte Einflußnahmen zu vermeiden.

Ausschlaggebend ist dabei die möglichst minimale Beanspruchung des Funkkanals. Dies ist vor allem durch eine Minimierung der Anzahl der versendeten Nachrichten[33] durch *local logging* (das unten erläutert wird) zu erreichen. Jeder Sensorknoten verfügt aus diesem Grund schon vor Beginn der Ausführung des Testfalls über eine vorkonfigurierte Menge von sogenannten Steuerbefehlen. Solche Steuerbefehle können verschiedene Funktionen zur Testdurchführung durchführen, wie beispielsweise den Aufruf einer Prozedur, um ein Paket mittels eines zu testenden Routingprotokolls zu versenden bzw. auf den Empfang eines Pakets zu lauschen. Weitere Funktionen solcher Steuerbefehle können die Aktivierung entsprechender Prozeduren der Dienste von ScatterUnit sein, um deren Funktionalitäten zum Zwecke der Testdurchführung in Anspruch zu nehmen (wie z.B. bestimmte Dienste zu Simulation und Protokollierung).

Um die Testdurchführung auf Basis des Testfalls automatisiert auf Sensorebene auszuführen, werden die notwendigen Aktionen zur Bewerkstelligung des dem Test unterliegenden Testfalls sowie zur Inanspruchnahme der Testinfrastrukturdienste in entsprechende Steuerbefehle überführt. Solche Steuerbefehle werden, wie oben erwähnt, durch ein spezielles auf dem Sensorknoten laufendes Skript realisiert. Daher haben solche Testskripte vor allem die Aufgabe, die Ausführung des Testverlaufs zu steuern und zu protokollieren. Im Laufe der Arbeit wird auf abstrakter Mo-

[33] Dies sind Befehle jeglicher Art: zur Initialisierung, Protokollierung, etc. Normalerweise werden solche Nachrichten ständig durch einen herkömmlichen zentralistischen polling-Ansatz zwischen den Knoten und der Sinke hin und her verschickt.

dellierungsebene von *Aktionen* (Actions) des Tests gesprochen. Auf Implementierungsebene wird der Begriff *Steuerbefehl* verwendet. Somit stellt ein Steuerbefehl auf Codierungsebene die Realisierung einer im formalen Modell modellierten Aktion dar. Beispielsweise ist das Versenden eines Pakets auf Modellebene eine Aktion. Beim dazu nötigen Steuerbefehl handelt es sich um den Prozeduraufruf des zugrunde liegenden Routingprotokolls bzw. der zu testenden Anwendung, d.h. im Allgemeinen ein Aufruf einer Funktionalität des Testobjekts oder eines Dienstes der Scatter-Unit-Schicht.

Des Weiteren werden die für den Test protokollierten Daten (z.B. stattgefundene Aktionen des Tests) erst nach Beendigung der Ausführung des Testverlaufs einmalig an die Senke verschickt. Dies führt zu einer weiteren Reduktion der ständigen wechselseitigen Inanspruchnahme des Funkkanals. „The problem of sending information over channels has been elegantly solved by local logging and transmission of the information once the test has completed.“[34]

Der Leitfaden zur Implementierung eines Testfalls liegt daher in der dezentralen Ausführung des zum Test gewählten Testszenarios. Somit setzt sich ein Testszenario in ScatterUnit aus einer Menge von Aktionen zusammen (z.B. das Versenden eines Pakets) sowie aus Ereignissen, deren Vorkommen man für den Fall erwartet, dass die Ausführung der Anwendung fehlerfrei vonstatten geht (z.B. der erfolgreiche Empfang eines Pakets durch den richtigen Zielknoten).

Daher besteht die Grundlage für die Evaluierung der Testfallausführung in der Erstellung einer Sequenz von Aktionen bzw. Ereignissen, die aus der Protokollierung des nebenläufigen Testverlaufs konstruiert werden. Aufgrund der nebenläufigen Ausführung muss weiterhin die Reihenfolge spezifiziert werden, in der die Aktionen ausgeführt werden sollen. Diese Ordnung ist dem Testszenario zu entnehmen, das, wie eingangs erwähnt, als Leitfaden des Testfalls fungiert. Das Schreiben des dazu nötigen Codes mit der Einhaltung der richtigen Reihenfolge der nebenläufig ablaufenden Aktionen ist jedoch ein schwieriges Unterfangen. Dies wird im kommenden Abschnitt anhand eines konkreten Anwendungsbeispiels demonstriert.

3.3. Die Implementierung des Testfalls in ScatterUnit

ScatterUnit benötigt für die Implementierung eines Testfalls wie bereits erwähnt sogenannte Testskripte, welche die für den Test nötigen Steuerbefehle implementieren. Daher muss der Entwickler bzw. Testfalldesigner sich mit den folgenden beiden Fragen befassen:

- Welche Steuerbefehle werden auf welchem Sensorknoten ausgeführt, für die das Testskript implementiert werden muss?
- Nach welchem Steuerbefehl wird ein weiterer Steuerbefehl ausgeführt?[35]

Um den Einsatz von ScatterUnit ausführlicher zu erörtern, wird im Folgenden die Implementierung eines konkreten Testfalls zum Testen des DSDV-Routingprotokolls [Perk94] mit Hilfe von ScatterUnit demonstriert. Die Auswahl eines Routingprotokolls als Testobjekt für die Demonstration der Funktionalität von

[34] Ein (anonymer) Reviewer des Visual ScatterUnit Papers in ACM/IEEE Models08

[35] Bei der Implementierung eines Steuerbefehls für das Testskript eines Sensorknotens muss zur Beantwortung der zweiten Frage in Betracht gezogen werden, dass die vorherige Aktion möglicherweise auf einem anderen Sensorknoten ausgeführt wurde.

Visual ScatterUnit wurde bewusst getroffen, da Routing in drahtlosen Sensornetzen einen gut erforschten und repräsentativen Bereich darstellt. Im Kontext der Entwicklung von Routingprotokollen ist man (im Allgemeinen, aber besonders auch im Bereich drahtloser Sensornetze) auf Simulationswerkzeuge angewiesen, um das Verhalten des Netzes hinsichtlich des entwickelten Protokolls zu testen. Daher ist die Entwicklung solcher Simulationswerkzeuge ein Bereich, in dem viel geforscht wird. Geht man jedoch einen Schritt zurück, so muss zuerst gewährleistet werden, dass die Implementierung des Routingprotokolls getestet werden kann und zwar auf den sich bereits im Betrieb befindlichen Sensorknoten. An dieser Stelle mangelt es an entsprechenden Werkzeugen, was die Notwendigkeit von Visual ScatterUnit unterstreicht.

Insbesondere im Bereich drahtloser Sensornetze müssen Routingprotokolle auf Topologieänderungen eingestellt sein, um diese robust zu bewältigen. Ein einfaches und zugleich repräsentatives Testszenario für ein solches Routingprotokoll ist der Aufbau eines kleinen multi-hop Netzwerkes, in dem anhand des Routingprotokolls die Datenpakete von einem Knoten an einen oder mehrere benachbarte Knoten weitergereicht werden. Wie zuvor dargestellt, besteht die Hauptaufgabe des Testfalls im Versenden eines designierten Pakets auf Basis des zu testenden Routingprotokolls und die gleichzeitige Beobachtung, ob das Paket den Zielknoten ordnungsgemäß erreicht oder nicht. Bei der Ausführung des Testfalls kann ein Fehler auftauchen, wenn das Routingprotokoll bei der Zustellung der Datenpakete versagt, weil die Pakete unterwegs nicht korrekt weitergeleitet wurden.

Um nun nachvollziehen zu können, an welcher Stelle (Softwaremodul bzw. Funktionalität) das Versagen aufgetreten ist, muss der Testprozess in der Lage sein, die Route des betreffenden Pakets zu rekonstruieren. Dazu werden die Aktionen, die zur Weiterleitung des Pakets von den benachbarten Knoten stattgefunden haben, in der korrekten Reihenfolge benötigt. Man sieht, dass sich das Testobjekt auf mehrere miteinander agierende Knoten erstreckt. ScatterUnit verfügt daher über einen Dienst zur Koordination der sensorübergreifenden Aktionen. [Rafi03]

Wie sieht nun das korrespondierende Testszenario in ScatterUnit konkret aus? Zu Beginn wird das drahtlose Sensornetz mit der in Abbildung 19 (links) dargestellten Topologie konfiguriert. Zunächst sendet Sensorknoten 1 ein Paket an Sensorknoten 4. Sensorknoten 4 empfängt das von Sensorknoten 1 versendete Paket. Danach ändert sich die Topologie des Netzes (vgl. Abbildung 19, rechts), so dass sich auch der Pfad zwischen den beiden Sensorknoten 1 (Sender) und 4 (Empfänger) ändert. Dabei bewegt sich Sensorknoten 4 außerhalb der Reichweite von Sensorknoten 3 und tritt in die Reichweite des Sensorknotens 2 ein. Eine solche Topologieänderung zu Testzwecken kann mittels einer entsprechenden Paketfilterung mit Hilfe eines speziellen Dienstes von ScatterUnit bewerkstelligt werden. Nun sendet Sensorknoten 1 ein weiteres Paket an Sensorknoten 4 und Sensorknoten 4 empfängt das Paket erneut.

Abbildung 19 Änderung der Topologie während der Testdurchführung [Kame08]

Aufgrund der Topologieänderung des Netzes muss das Routingprotokoll beim Versenden des zweiten Pakets von Knoten 1 zu 4 einen anderen Pfad zwischen den beiden betroffenen Knoten finden. Falls Sensorknoten 4 das zweite Paket nicht empfängt, bedeutet dies, dass das Routingprotokoll die Topologieänderung nicht abgefangen hat, was zu einem Fehler im Testverlauf führt.

Der Testfall wurde durch vier Testskripte implementiert. Zuerst ruft ScatterUnit kurz vor der Testausführung eine Initialisierungsmethode (set-up Methode) dieser Testskripte auf. Dadurch werden verschiedene, für den Testverlauf notwendige Dienste[36] von ScatterUnit aktiviert. Sobald Sensorknoten 4 von 1 das Datenpaket empfangen und dem eigenen Skript diese Aktion mitgeteilt hat, sendet Sensorknoten 1 nach der Topologieänderung das zweite Paket. Schlägt das Versenden von einem der Datenpakete fehl (beispielsweise nach time out), so wird die Ausführung des Testfalls abgebrochen. Die Logs aller am Testfall beteiligten Sensorknoten werden während der Ausführung des Testverlaufes akkumuliert und nach seiner Beendigung zur Analyse an den zentralen Rechner geschickt (auf dem das Rahmenwerk läuft), wo die verschiedenen Logs zu einem zeitlich geordneten kollektiven Log-Protokoll zusammengefügt werden. Ein solches Log enthält im Wesentlichen die folgenden Informationen (Aktionen des Testverlaufs) und zwar für den Fall, dass das Versenden des zweiten Pakets nach der Topologieänderung fehlgeschlagen ist:

1. Die Ausführung des Testfalls ist gestartet.
2. Sensorknoten 4 beginnt zu warten.
3. Sensorknoten 1 sendet das erste Datenpaket.
4. Sensorknoten 4 empfängt das erwartete Datenpaket von Sensorknoten 1.
5. Sensorknoten 4 verlässt die Reichweite des Sensorknotens 3.
6. Sensorknoten 4 begibt sich in die Reichweite des Sensorknotens 2.
7. Sensorknoten 4 geht in den Wartezustand (ähnlich wie in 2).
8. Sensorknoten 1 bricht die Ausführung des Testfalls nach time out ab.

Das Log wird durch verschiedene Prüfroutinen bearbeitet, wobei jede Prüfroutine für die Überprüfung bestimmter Arten von Fehlern zuständig ist. Somit wird eine Separierung von Belangen bei der Analyse und Auffindung der Fehler erreicht, was zu einer optimierten und strukturierten Bearbeitung des Logs führt. So meldet beispielsweise eine der Prüfroutinen, dass Sensorknoten 1 das zweite Datenpaket nicht erfolgreich versenden konnte. Das besondere dabei ist, dass diese Aktivität unmittelbar nach der Topologieänderung ausgeführt wurde. Um die korrekte Ordnung der dezentralen Ausführung der Aktionen zu gewährleisten, wird ein spezieller Dienst in ScatterUnit zur Verfügung gestellt, nämlich der Befehle-Dienst (Command Service). Nach der Ausführung einer Aktion auf einem Sensorknoten wird ein dazu designierter Befehl (Signalisierung) an denjenigen Sensorknoten verschickt, auf dem die nächste Aktion ausgeführt wird.

Solche Signalisierungen fungieren als Markierungen zur Koordination des auf verschiedene Knoten verteilten Testverlaufs. Auf der Grundlage solcher Signalisierungsbefehle kann nach der Beendigung des Tests ein kohärentes Log konstruiert

[36] Eine Auflistung der unterschiedlichen Dienste von ScatterUnit findet sich in 4.4, in dem die technische Realisierung der Testautomatisierungsschicht ScatterUnit erläutert wird.

werden, das den verteilten Testablauf bezüglich aller im Netzwerk beteiligten Sensorknoten als geschlossenes und geordnetes Gefüge repräsentiert.

Dieser Vorgang ist offensichtlich eine recht komplexe Aufgabe, insbesondere wenn man umfassende Tests durchführen will. Folglich erschwert sich für den Entwickler die Implementierung des Testszenarios (das als Leitfaden des gesamten Testverlaufs fungiert) erheblich. Ist man in der Lage, diesen komplexen Vorgang dem Entwickler durch Automatisierung abzunehmen und ihn an einen Codegenerator zu delegieren, erleichtert dies die Implementierung des Testszenarios erheblich – wobei alle mit ScatterUnit erreichten Vorteile erhalten bleiben. An dieser Stelle setzt das modellgetriebene Paradigma an, was im kommenden Abschnitt näher behandelt wird. Dabei muss sich der Entwickler lediglich auf den Entwurf des Leitfadens konzentrieren – also das passende Modell des Testszenarios erstellen, dessen Implementierung weitestgehend automatisiert vonstatten geht. Wie dies zu realisieren ist, wird im folgenden Abschnitt genauer erläutert.

3.4. Modellgetriebenes Testen in Visual ScatterUnit

Das Hauptanliegen beim modellgetrieben Testen in Visual ScatterUnit ist die formale Modellierung des Testfalls, um daraus den Code für die von ScatterUnit benötigten Testskripte automatisiert zu generieren. Daher liegt der zentrale Punkt des mit Visual ScatterUnit verfolgten modellgetriebenen Ansatzes in der Bestimmung der Schlüsselaspekte des Testfalls, welcher in einer abstrakten Weise modelliert wird. Ein solches formales Modell stellt eine direkte Repräsentation des Testszenarios dar, in dem die korrespondierenden Aktionen, deren Zuweisung zu den jeweiligen Sensorknoten sowie die Festlegung ihrer Ausführungsreihenfolge eindeutig durch das Modell spezifiziert werden müssen. Denn auf Basis der durch das Modell vorgegebenen Informationen kann der Codegenerator automatisiert mittels der von ihm generierten Testskripte und den darin enthaltenen Signalisierungsbefehlen (gemäß des im letzten Abschnitt dargestellten *local logging*) ein Log für den gesamten verteilten Testverlauf erstellen. Dies ermöglicht die Evaluierung des Tests für alle beteiligten Knoten.

Bei der Modellierung eines Testfalls in Visual ScatterUnit ist der Ausgangspunkt, wie erwähnt, ein Aktivitätsdiagramm. Ein Beispiel eines solchen Diagramms zeigt Abbildung 20, wobei das Diagramm sich auf das im letzten Abschnitt diskutierte Testszenario des DSDV-Routingprotokolls bezieht. Die Notation des formalen Modells ähnelt stark den UML-Aktivitätsdiagrammen. Der Hauptunterschied betrifft die Rolle der Aktivitäten innerhalb des Diagramms: Eine Aktivität repräsentiert eine Gruppe von Aktionen, die einem bestimmten Zweck dienen, wie z.B. der Änderung der Topologie des Netzes. Desweiteren wird in der internen Darstellung einer Diagrammaktivität spezifiziert, auf welchem Sensorknoten die repräsentierten Aktionen ausgeführt werden. Daher sollte das in Abbildung 20 gezeigte Diagramm wie folgt gelesen werden: Ist der Testfall gestartet, werden zwei Aktivitäten nebenläufig ausgeführt. Sensorknoten 1 versendet das erste Datenpaket und Sensorknoten 4 wartet auf seinen Empfang. Nachdem das Paket empfangen wurde, ändert sich die Topologie des Netzes. Das zweite Paket wird nun vom Sensorknoten 1 versendet und Sensorknoten 4 wartet auf seinen Empfang. Wird das Paket erfolgreich von Zielknoten 4 empfangen, terminiert der Testfall.

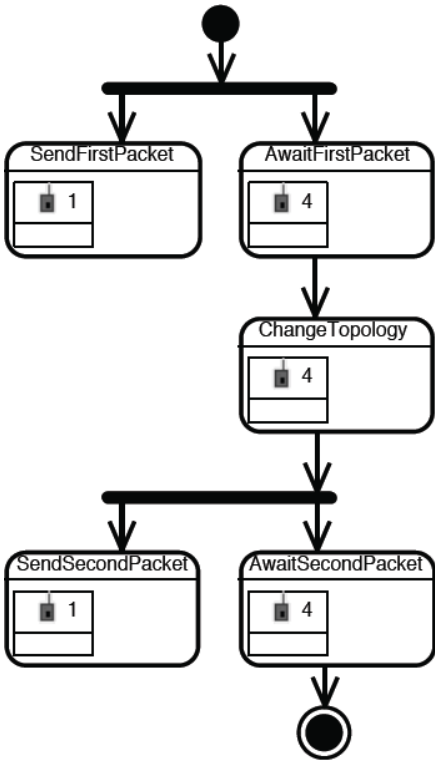


Abbildung 20 Aktivitätsdiagramm zur Modellierung des Verlaufs des Testszenarios

Dieses Diagramm dient der Darstellung der intuitiven Art der Repräsentation des Testszenarios. Um jedoch in der Lage zu sein, aus dem besagten Modelldiagramm den entsprechenden Code der Testskripte zu generieren, muss sich der Entwickler auch mit den Feinheiten befassen. Daher wird zu jeder im Hauptaktivitätsdiagramm modellierten Aktivität ein zusätzliches, detaillierteres Diagramm konstruiert, um diejenigen Aktionen feinkörniger zu modellieren, aus denen die Aktivität besteht. Ein Beispiel dafür zeigt das Diagramm in Abbildung 21, das die Aktivität `Change Topology` des Aktivitätsdiagramms in Abbildung 20 detaillierter darstellt:

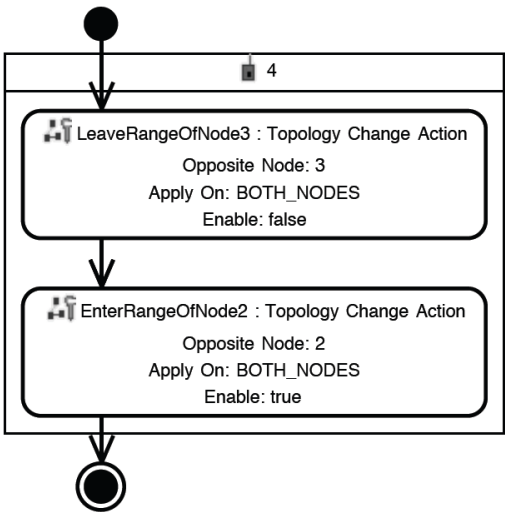


Abbildung 21 Das verfeinerte Diagramm der Aktivität "Change Topology" aus Abbildung 20

Das Diagramm zeigt zwei Aktionen: Die erste Aktion bezieht sich auf den Austritt des Sensorknotens 4 aus der Reichweite des Sensorknotens 3 und die zweite auf den Eintritt des Knotens 4 in die Reichweite des Knotens 2. Dies sind die beiden Aktionen, welche die Topologieänderung zur Folge haben. Da die Aktionen sich innerhalb des Aktivitätsknotens von Sensorknoten 4 im Diagramm befinden, werden sie diesem Knoten zugewiesen. Infolgedessen werden die dazu benötigten Codeartefakte als Teil der Testskripte von Sensorknoten 4 generiert.

Allerdings wird, wie bereits erwähnt, nicht immer der für die Testskripte zuständige Code ausschließlich aus dem Diagramm des Testfalls vollständig generiert. In der Regel sind bestimmte Aktionen manuell zu implementieren. Ein Beispiel einer solchen Aktion sieht man im Diagramm der folgenden Abbildung 22. Das Diagramm ist zuständig für die Aktivität `SendSecondPacket`, die im Aktivitätsdiagramm des Testfalls (vgl. Abbildung 20) modelliert ist. Im Diagramm ist nur eine einzige Aktion enthalten. Dies liegt daran, dass die übrigen Aktionen von manuell implementiertem Code ausgeführt werden. Um ein Datenpaket über den Funkkanal mit Hilfe des Routingprotokolls – die zu testende Anwendung – zu senden, muss das Paket zuvor erstellt und anschließend die entsprechende Methode des Routingprotokolls aufgerufen werden. Eine solche Aktion ist eng an die zu testende Anwendung gebunden. Solche anwendungsspezifischen Aktionen werden daher vom Entwickler manuell implementiert. Es wäre nicht sinnvoll, jegliche mögliche Aktion zu modellieren, um daraus den zugehörigen Code zu generieren, da ein solcher Code nicht über einen statischen Charakter verfügt. Solche feinkörnigen Codesegmente richten sich stark nach der zu testenden Anwendung, und weil das Spektrum der in ScatterUnit zu testenden Anwendungen uneingeschränkt bleiben sollte, werden solche anwendungsspezifischen Aktionen dem Entwickler überlassen, um diese dann nach seiner Intention maßgeschneidert umzusetzen. Der hier verfolgte hybride Ansatz verleiht dem Rahmenwerk einen hohen Grad an Offenheit bezüglich der potentiell zu testenden WSN-Anwendungen. Dieser Ansatz wurde durch den angeführten Testfall illustriert: „The authors use an example test case to illustrate the approach and improve understanding of the reader. […] The explanation helps to understand where the manual step is involved in the test case generation. Also the authors provide a good reason as to why this manual code is needed.“[37]

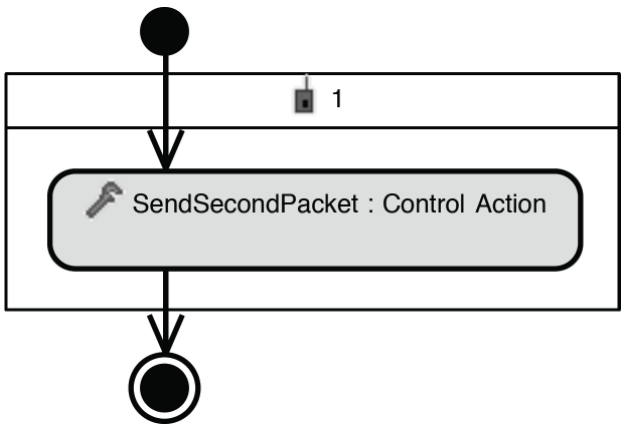


Abbildung 22 Verfeinertes Diagramm der Aktivität "SendSecondPaket"

[37] Ein (anonymer) Reviewer des Papers Visual ScatterUnit in ACM/IEEE Models´08

Abbildung 23 zeigt ein Diagramm, das die Aktionen der Aktivität `AwaitSecondPacket` beinhaltet: Sobald das erwartete Paket empfangen wurde, wird der dafür in ScatterUnit zuständige Dienst abgefragt um die entsprechende Benachrichtigung abzuschicken. Diese Benachrichtigung wird anhand eines Ereignisses namens `SecondPacketReceived` repräsentiert. Dann folgt die Aktion `AbortIfTimeOut`, für die der Code manuell implementiert werden muss, um die Ausführung des Testfalls abzubrechen, wenn das Datenpaket nicht empfangen wurde. Manuell zu implementierende Aktionen werden im Diagramm durch eine graue Schattierung gekennzeichnet. Wenn das Paket empfangen wird und somit kein Fehler diesbezüglich in der Anwendung aufgetreten ist, wird dessen Inhalt anhand der Aktion `LogRadioPacket` geloggt. Mit der Log-Marke (Log Label) `SecondRadioPacket` wird auf die Aktion Bezug genommen und so kann die Integrität der Aktion mit der Prüfroutine (Assertion) `CheckPacketIntegrity` überprüft werden.

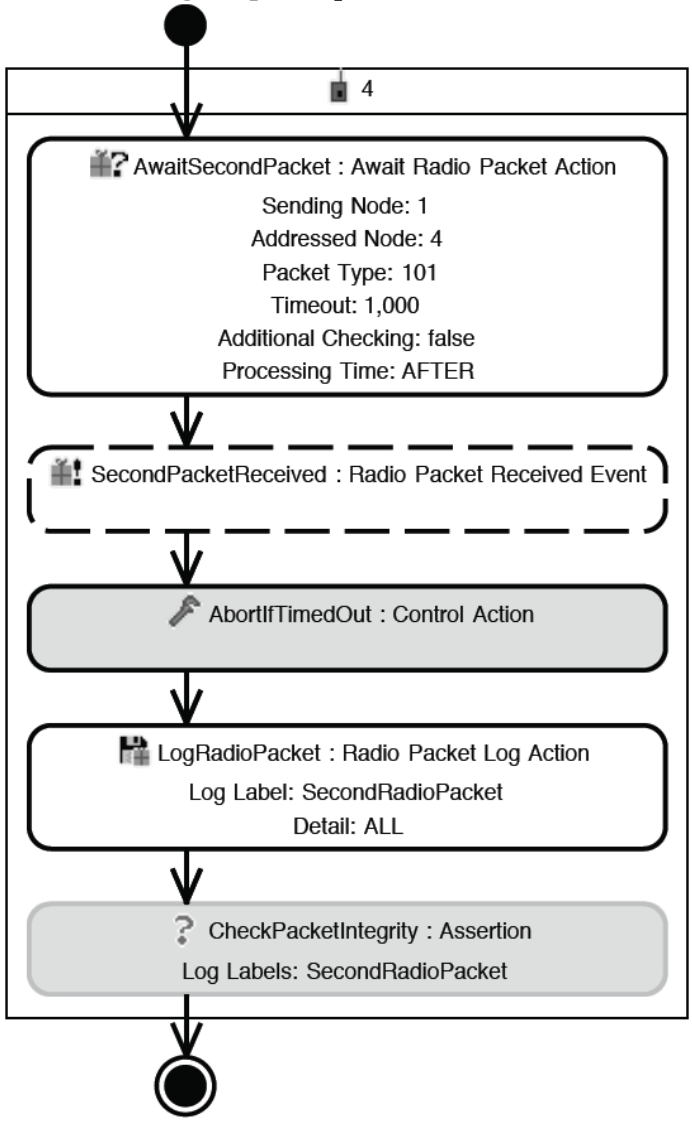


Abbildung 23 Verfeinerungsdiagramm der Aktivität „AwaitSecondPaket“

Die Testinformationen werden nach der Beendigung des Testfalls anhand des während der Testausführung akkumulierten Logs bei der zuständigen Prüfroutine bzw. dem zuständigen Dienst von ScatterUnit analysiert. Wie man unschwer erkennen kann, sind solche Prüfroutinen sehr anwendungsspezifisch, daher werden sie ebenfalls vom Entwickler manuell implementiert. So handelt es sich beispielsweise hier um die Überprüfung der Integrität des Pakets durch welche festgestellt wird, ob dessen Nutzlast während der Übertragung erhalten geblieben ist oder beschädigt wurde.

Bei der automatisierten Generierung der Testskripte eines Testfalls zerlegt der Codegenerator dieses in die darin enthaltenen Aktionen und weist diese dem Testskript des jeweiligen Sensorknotens zu, auf dem das Skript mitsamt seiner Aktionen ausgeführt wird. Um die Reihenfolge der Ausführung der Aktionen beizubehalten, werden, wie zuvor erläutert, die entsprechenden Signalisierungsbefehle des Testprotokollierungsprozesses von ScatterUnit eingesetzt. Da mit Hilfe der modellgetriebenen Visual ScatterUnit die Ausführungsreihenfolge der Aktionen anhand der Richtungspfeile des Aktivitätsdiagramms modellierbar ist, wird auf dieser Basis der Codegenerator in die Lage versetzt zu unterscheiden, ob zur Generierung der betroffenen Aktionen entsprechende Signalisierungsbefehle erstellt werden müssen oder nicht. Dies ist möglich, da Aktionen, die mit einem Richtungspfeil im Modelldiagramm verbunden sind, zugleich in eben diesem Modelldiagramm ihren verschiedenen Sensoren eindeutig zugeordnet werden. Wie man hier sehen kann, wird die recht komplexe Aufgabe der technischen Realisierung der koordinierten Ausführung der Aktionen vom Codegenerator übernommen.

Der für die Bewerkstelligung der Ausführungsanordnung automatisiert generierte Code ist das Herzstück des Infrastrukturcodes der unterliegenden Domäne (in diesem Zusammenhang der automatisierte Test der ScatterWeb-Anwendungen). Denn dieser Code muss in Anspruch genommen werden, um Testfälle mit ScatterUnit zu erstellen. Demzufolge kann der Anwendungsentwickler bzw. Tester den Fokus auf den Entwurf des Testszenarios richten, um den Testverlauf zu optimieren, während die komplexen technischen Details (Erstellung des Infrastrukturcodes) an den Codegenerator delegiert werden. Neben diesem Infrastrukturcode zur Koordinierung des Testverlaufs ist es in Visual ScatterUnit möglich, den der Evaluierung der Testresultate korrespondierenden Infrastrukturcode ebenfalls automatisiert zu generieren. Dies ermöglicht einen visuellen automatisierten Debuggingprozess der ScatterWeb-Anwendungen.

3.5. Der visuelle automatisierte Debuggingprozess

Nach der Terminierung der Ausführung des Testfalls steht ein Log zur Verfügung, das sämtliche relevante Ereignisse enthält, die während des Testablaufs stattgefunden haben. Anschließend wird dieses Log von verschiedenen Prüfroutinen auf vorgekommene Fehler überprüft. Jede solche Routine ist für eine bestimmte Fehlerklasse zuständig. Beispielsweise überprüft die Prüfroutine `CheckPacketIntegrity` im Diagramm der vorherigen Abbildung 23 die Integrität der Nutzlast des versendeten Pakets. Für den Fall, dass die Nutzlast beschädigt wurde, wird dies von derjenigen Prüfroutine gemeldet, die die Überprüfung zur Evaluierung der geloggten Daten bearbeitet hat.

Um eine optimierte Prüfung spezifischer Fehler durch die korrespondierende Prüfroutine zu erreichen, müssen zwei Tätigkeiten ausgeführt werden: Zuerst wer-

den während der Ausführung des Testfalls die Testdaten als Aktionen geloggt, aus denen Rückschlüsse auf das Auftreten des gesuchten Fehlers gezogen werden können. Daher wird das Testfallmodell um entsprechende Log-Aktionen gemäß dem jeweiligen unterliegenden Testszenario vom Entwickler erweitert. Zweitens werden die gesammelten Informationen anhand dieser Logs evaluiert. Daher werden die relevanten Logeinträge aus dem Logstrom ausgesucht, um sie bei der entsprechenden Routine zu analysieren. Um dies technisch zu bewerkstelligen, müssen die Logeinträge entsprechend den Anforderungen der Prüfroutinen aufbereitet werden. Zu diesem Zweck wird das Testfallmodel zusätzlich zu den Logaktionen mit den korrespondierenden Markierungen (log labels) versehen. So gibt die Aktion `SecondRadioPacket` im Testfallmodell die Logmarkierung `LogRadioPacket` an, wobei diese Logmarkierung gleichzeitig auf die für die Überprüfung der Aktion `SecondRadioPacket` zuständige Prüfroutine `CheckPacketIntegrity` verweist. Der Prüfroutine stehen somit sämtliche Daten zur Verfügung, so dass die technische Realisierung der gerade beschriebenen Abläufe ebenfalls voll automatisiert durch den Codegenerator vonstatten gehen kann.

Obwohl der Code für die Selektierung der Logeinträge automatisiert generiert werden könnte, muss er dennoch der Anwendungslogik einer Prüfroutine entsprechend manuell implementiert werden. Der Grund hierfür liegt vor allem darin, dass die Routine zur Überprüfung eines Fehlers ebenfalls anwendungsspezifisch ist. Es gibt jedoch einen typischen Fehler, dessen Code automatisiert generiert werden kann. Dieser Fehlertyp geht von der folgenden Annahme bezüglich des Testfallmodells aus: Die modellierte Sequenz der Aktionen im Diagramm richtet sich danach, ob der Testverlauf am Ende eine fehlerfreie Anwendungsausführung aufweist. Wird die Ausführung unterbrochen, so impliziert dies eindeutig, dass ein Fehler während der Anwendungsausführung aufgetreten ist. Es wäre beispielsweise möglich, dass die Aktion `SendSecondPacket` die Ausführung des Testfalls abbricht, um darauf hinzuweisen, dass das Versenden des Pakets mittels des zu testenden Routingprotokolls fehlgeschlagen ist. Eine dazu generierte Prüfroutine wird nun über das Auftreten des Fehlers berichten, indem die betroffene Aktion indiziert wird. Für den Abbruch der Ausführung wird kein manuell geschriebener Code benötigt, lediglich ein einziger Systemaufruf, der diesen veranlasst. Fehler, deren Vorkommen auf diesem Weg berichtet wird, reduzieren die Implementierungszeit für die Fehlerbenachrichtigung, weil auf sie prompt mit Abbruch reagiert wird.

Alles in allem werden die folgenden Testresultate analysiert: Fehler aus benutzerdefinierten Log-Aktionen, die durch deren korrespondierende Prüfroutinen überprüft werden; Fehler, die durch den Abbruch der Ausführung des Testfalls verursacht werden, und die Daten, die durch die Protokollierung des Testverlaufs als Aktionen geloggt werden. Um diese Resultate auf einfache Art hinsichtlich des Anwendungsverhaltens auslesen bzw. interpretieren zu können, fließen sie in das Diagrammmodell des Testfalls ein. Abbildung 24 zeigt die Visualisierung der Fehlermeldung, die durch die Prüfroutine `CheckPacketIntegrity` aus Abbildung 23 verursacht wurde. Der Fehler wird durch das entsprechende Blitz-Icon im Diagramm angezeigt. Der Tooltip des Icons blendet dann die folgende Meldung ein: „The payload of the data Paket was corrupted“. Außerdem gelangt man durch das i-Icon der Aktion `LogRadioPacket` im Diagramm zu den Daten des betroffenen Pakets.

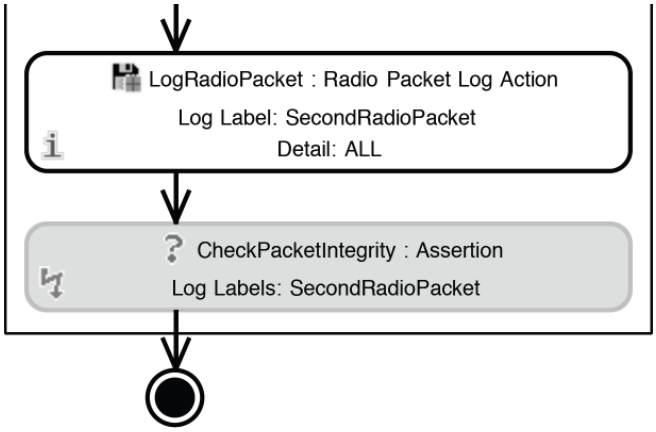


Abbildung 24 Visualisierung der Fehlermeldung im Modell durch den Einbau von entsprechenden Icons Links der Aktionen

Damit sind die für den Test relevanten Informationen für den Benutzer leicht handhabbar, so dass er auf übersichtliche Weise zwischen den Diagrammen zwecks Informationsgewinnung zum Ausgang des Testverlaufs navigieren kann. Ohne die Visualisierung der Fehlermeldungen durch die Verbindung zwischen den aufgetretenen Fehlern und deren korrespondierenden Modellkonstrukten im Testfalldiagramm müsste der Benutzer, um die protokollierten Informationen des Testverlaufs nachvollziehen zu können, die Einträge des textuellen Logs selbst in den Kontext des Testszenarios einbetten, was eine recht komplexe und zeitaufwändige Tätigkeit wäre.

Die optimierte Lesbarkeit der Testresultate vereinfacht ebenfalls den Einsatz von Log-Aktionen, um die Nachvollziehbarkeit der Fehlermeldungen zu verbessern. Erhält man die Testresultate wie in Abbildung 24 gezeigt, stellt man möglicherweise bei der Begutachtung der für die Aktion `LogRadioPacket` geloggten Daten fest, dass die Nutzlast abgesehen vom letzten Byte, das bei der Übertragung verloren gegangen ist, unbeschädigt ist. Dies führt beim Benutzer zu der Vermutung, dass beim Versenden des Pakets die Länge der Nutzlast möglicherweise falsch angegeben wurde. Er erhält so einen Anhaltspunkt, wo er im Code des Routingprotokolls nach der Fehlerursache suchen kann. Dieses Beispiel zeigt, wie man im Modell flexibel anhand der benutzerdefinierten Log-Aktionen navigiert, um die Fehler im Code der zu testenden Anwendung zu lokalisieren.

Nachdem ein Fehler nach der Ausführung des Testfalls erkannt wurde, ist der folgende Schritt die Korrektur des Fehlers. Bevor jedoch die betroffene Codestelle korrigiert werden kann, muss die Ursache der Fehlerentstehung innerhalb des Codes lokalisiert werden. Dies wird im Allgemeinen bewerkstelligt, indem die Codestelle fortschreitend isoliert wird, bis der Fehler zielgenau lokalisiert wurde. Der Ablauf dieser Aktivität wird in Abbildung 25 veranschaulicht. Agans in [Agan02] empfiehlt an dieser Stelle einen Teile-und-Herrsche-Ansatz. [Corm90]

Um das oben beschriebene Vorgehen zu illustrieren, betrachten wir das folgende, im Kontext des Routings übliche Szenario: Angenommen der Testfall entdeckt einen Fehler in der Anwendung, der darin besteht, dass nicht alle Sensordaten zur Senke korrekt übertragen wurden. Die Ursache kann möglicherweise auf ein Versagen bei der Erfassung der Umgebungsdaten der jeweiligen Sensorknoten zurückzuführen sein. Eine weitere mögliche Ursache liegt in der Übertragung der Daten. Um nun

festzustellen, welche der beiden Möglichkeiten die Ursache war, muss zunächst überprüft werden, ob die vom Sensorknoten erfassten Daten im Sensorknoten selbst vollständig gespeichert wurden. Ist dies der Fall, muss der zweiten Möglichkeit nachgegangen werden, indem überprüft wird, ob eine Beeinträchtigung bei der Datenübermittlung zur Senke vorliegt. Angenommen es handelt sich bei der Fehlerursache um die erste Möglichkeit, so wurde derjenige Teil der Anwendung identifiziert, auf den das Versagen zurückzuführen ist. Dies ist dasjenige Segment des Anwendungscodes, das für die Datenerfassung bzw. Speicherung der aufgenommenen Daten zuständig ist.

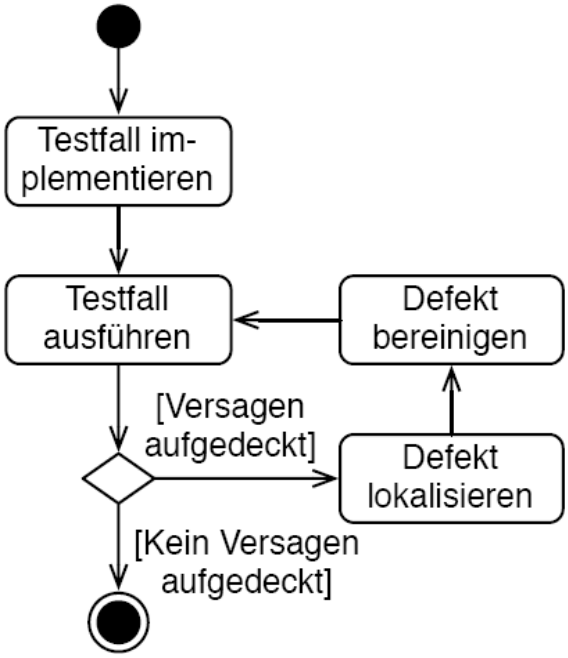


Abbildung 25 Iterative Vorgehensweise bis der Fehler im Code aufgefunden wurde [Kame08]

Nun kann der Fokus noch feinkörniger gelegt werden, indem die Fehlerlokalisierung weiter geteilt und überprüft wird, ob es an dem Modul der Speicherung oder dem/denen zur Datenerfassung liegt.[38] Diese Vorgehensweise wird nun sukzessiv betrieben, bis schließlich die Fehlerursache im Code lokalisiert wurde.

Der Prozess der Fehlerisolierung im Implementierungscode der Anwendung ist eine recht anspruchsvolle kognitive Aktivität, die stark auf den Annahmen des Testers beruht. Der Tester stellt Hypothesen bezüglich des Fehlers auf und verifiziert diese iterativ. [Xu04] Danach wird entsprechend dem Teile–und–Herrsche–Ansatz vorgegangen, wobei bei jeder vom Tester aufgestellten Hypothese eine Iterationsstufe im Verfahren entsteht, nach der der Code weiter zerlegt und somit der Problembereich kleiner wird. Im obigen Beispiel wäre es möglich, die Annahme zu treffen, dass der Fehler auf die Speicherungsfunktionalität der gesammelten Daten zurückzuführen ist. Um diese Annahme zu verifizieren, müssen die relevanten, vom Sensorknoten gesammelten Daten in Betracht gezogen werden. Dieser Schritt ist für die

[38] Um die Sensorik des Knotens zu überprüfen, kann parallel die statistische Analyse von ScatterPlug eingesetzt werden (mehr dazu in 6.4). Dies ist ein Beispiel für die ergänzende Zusammenarbeit der verschiedenen Features bzw. Komponenten der ScatterClipse Plattform.

Durchführung des oben beschriebenen Prozesses unerlässlich. An dieser Stelle stellt sich die Frage, wie das Gesamtbild des Prozesses zur Isolierung der Fehler innerhalb des Codes genau aussieht. Beim Einsatz der modellgetriebenen Visual Scatter-Unit geschieht dies wie in Abbildung 26 dargestellt:

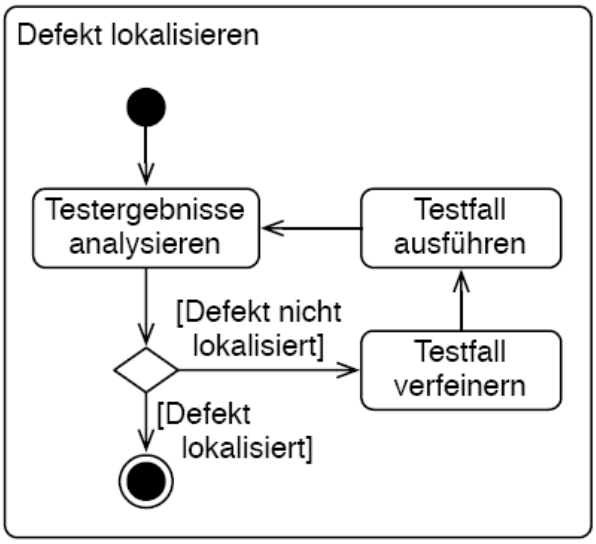


Abbildung 26 Der Prozess zur Isolierung von Codefehlern innerhalb der zu testenden Anwendung [Kame08]

Zuerst werden, wie erwähnt, die in das Modell inkorporierten Testresultate analysiert. Auf Basis der daraus erhaltenen Rückschlüsse auf das Verhalten der Anwendungen werden neue Hypothesen aufgestellt, die ihrerseits festlegen, wie weiter vorgegangen wird. Das Modell des Testfalls wird entsprechend verfeinert und der Testfall erneut ausgeführt. Auf diese Weise wird sukzessiv verfahren, bis das Modell vollständig von Fehlern bereinigt und somit die Versagensursache endlich gefunden wurde. Um dies zu veranschaulichen, kehren wir zum Testfall des DSDV-Routingprotokolls zurück und versuchen, das eben Gesagte an diesem Beispiel zu demonstrieren.

Die Überprüfung der Testresultate nach der Ausführung des Testfalls bei der ersten Iteration zeigt, dass das zweite Datenpaket von Sensorknoten 4 nicht empfangen wurde. Wir stellen daher die Hypothese auf, dass das Routingprotokoll das Paket über eine falsche Route weitergeleitet hat. Nun müssen wir bestimmen, über welche Route das Paket fälschlicherweise weitergeleitet wurde. Das Testfalldiagramm wird verfeinert (vgl. Abbildung 27), indem es um weitere Aktivitäten erweitert wird, durch die der unmittelbar benachbarte Knoten, zu dem das betroffene Paket weitergeleitet wurde, festgestellt werden soll.

Im Fall, dass das Paket über die restlichen Sensorknoten 2 bzw. 3 (1 ist der Sender und 4 ist der Empfänger) weitergeleitet wird, warten die Aktivitäten `AssertForwarding` und `AssertNoForwarding` auf das Datenpaket. Eine Prüfroutine wird dem Sensorknoten 2 zugewiesen, um einen Fehler zu registrieren, falls das Datenpaket nicht zu diesem Sensor weitergeleitet wird. Dieses Vorgehen ist dadurch begründet, dass vom Routingprotokoll nach der Topologieänderung erwartet wird,

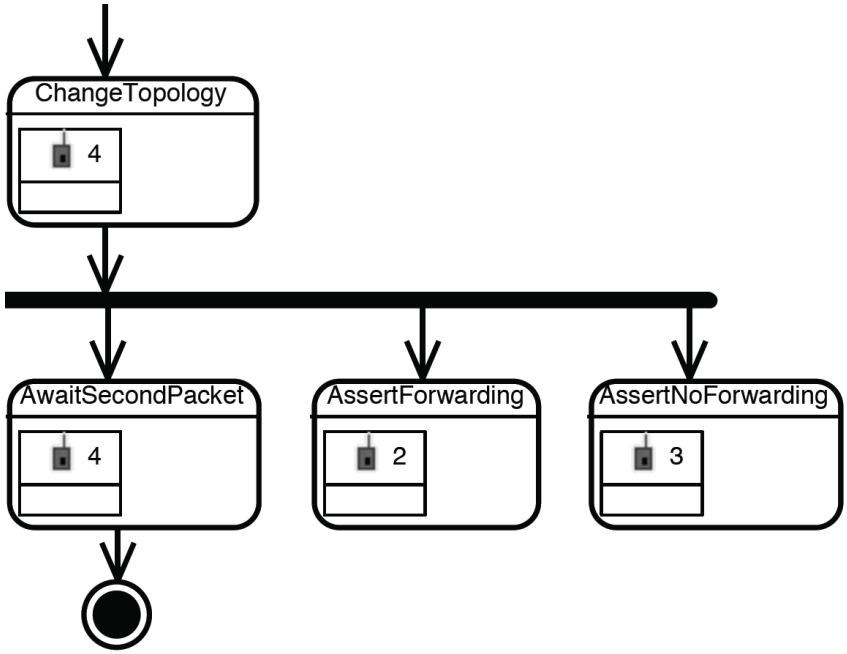


Abbildung 27 Das verfeinerte Diagramm des Testfalls [Kame08]

das Paket über diesen Sensorknoten weiterzuleiten. Die dem Sensorknoten 3 zugewiesene Prüfroutine bewirkt das Gegenteil: Hier wird ein Fehler berichtet in dem Fall, dass das Paket über diesen Sensorknoten weitergeleitet wird.

Nach der Ausführung und der daran anschließenden Überprüfung des verfeinerten Testfalldiagramms, wird möglicherweise festgestellt, dass das Paket wie erwartet durch den benachbarten Sensorknoten 2 weitergeleitet wurde. In diesem Fall muss die aufgestellte Hypothese revidiert werden. Eine Möglichkeit ist nun, dass dieser benachbarte Knoten nicht in der Lage war, das korrekt empfangene Paket seinerseits weiterzuleiten. Um diese neue Hypothese zu verifizieren, muss eine weitere Verfeinerung des Testfallmodells vorgenommen werden. Das entsprechende Diagramm wird um Aktionen erweitert, durch die die Routingtabelle von Sensorknoten 2 geloggt wird und zwar zu dem Zeitpunkt, in dem der Sensorknoten versucht, das betroffene Datenpaket weiterzuleiten.

Wenn daraufhin festgestellt wird, dass die Route zu Zielknoten 4 in der Tabelle nicht existiert, ist unsere Annahme zutreffend und hat uns auf die richtige Spur gebracht. Die Stelle der Fehlerursache im Code ist jedoch längst noch nicht lokalisiert worden. Nun muss die Verteilung der Einträge der Routingtabelle bzw. ihre Aktualität angesichts der Topologieänderung überprüft werden, weshalb eine weitere Verfeinerung mit neu hinzugefügten Aktionen zum Einsatz kommt, usw.

Dieses Beispiel zeigt, wie durch das kognitive Vorgehen bei der Fehlerlokalisierung mithilfe der iterativen Verfeinerung des Testfallmodells die Fehlerursache für das Versagen der zu testenden Anwendung lokalisiert werden kann. Die modellgetriebene Visual ScatterUnit vereinfacht diese Vorgehensweise, indem sie ermöglicht, die Testdiagramme zielgerichtet zu verfeinern, um die vom Softwaretester aufgestellten Hypothesen hinsichtlich der Fehlerlokalisierung zu verifizieren. Dabei liegt der große Nutzen des modellgetriebenen Paradigmas darin, dass es möglich ist, den in hohem Maße iterativen Verfeinerungsprozess des Modells zu automatisieren: Werden bei weiterer Verfeinerung neue Aktionen ins Testfalldiagramm eingebettet,

muss die Ordnung aller im Diagramm vorhandenen Aktionen entsprechend umgestaltet werden. Dies muss der entsprechende Code zur Koordination der Ausführung der Testskripte leisten. Da dieser Code mit Hilfe von Visual ScatterUnit automatisiert generiert wird, muss der Softwaretester die technische Umsetzung der Modellverfeinerung nicht selbst bewältigen. Das gleiche trifft auf den zur Evaluierung der Testresultate benötigten Infrastrukturcode zu: Die Integration der den Prüfroutinen zugewiesenen Logmarkierungen (Log labels) in das Testfalldiagramm ermöglicht die maßgeschneiderte Generierung der Codesegmente der zugehörigen Prüfroutinen zur Filterung der korrespondierenden Einträge im textuellen Log. Dieses wird durch *local logging* zur verteilten Protokollierung der Testfallausführung von ScatterUnit zusammengestellt. Wie man sieht, wird das Schreiben und bei sich wiederholender Verfeinerung des Modells nochmalige Schreiben des von ScatterUnit benötigten Infrastrukturcodes vollständig an den Codegenerator delegiert und geht automatisiert vonstatten. So wird der angestrebte Nutzen des modellgetriebenen architekturzentrierten Ansatzes beim Testen von Anwendungen drahtloser Sensornetze optimal erreicht.

3.6. Zusammenfassung

Um Anwendungen drahtloser Sensornetze adäquat testen zu können, sind entsprechende Dienste erforderlich, die beispielsweise die nötige Koordination des verteilten Testverlaufs sowie Simulationsfunktionalitäten bereitstellen. Solche Dienste werden auf automatisierte Art von der Testautomatisierungsschicht ScatterUnit angeboten. Die Dienste von ScatterUnit und deren Funktionsweise sowie die Erstellung eines Testfalls mit Hilfe von ScatterUnit wurden zu Beginn des Kapitels anhand eines konkreten Anwendungsszenarios, dem DSDV-Routingprotokoll ausführlich geschildert und diskutiert. Dennoch blieb die entsprechende Codierung eines Testfalls für eine auf den verschiedenen Knoten verteilt auszuführende Anwendung ein schwieriges Unterfangen. Hier war eine Steigerung der Automatisierung von Nöten, um diese Aufgabe so weit wie möglich an einen Codegenerator zu delegieren, damit der Anwendungsentwickler sich stärker der Konzeption und dem Entwurf des Testfalls widmen kann, anstatt den technischen Belangen. Hierzu wurde das modellgetriebene Visual ScatterUnit entwickelt, was im Abschnitt 4 vorgestellt wurde. Der Fokus des folgenden Abschnitts lag in der Analyse der protokollierten Testergebnisse nach Beendigung der Testausführung, um eine tiefere Einsicht in das Verhalten der zu testenden Anwendung zu erhalten. So wurde gezeigt, wie die – im Sinne der MDSD – ausführbaren formalen Modelle zweigleisig eingesetzt werden können, indem einerseits der Code für den Anwendungstest durch sie generiert werden kann, andererseits sie aber auch eingesetzt werden, um das Verhalten der zu testenden Anwendung zu visualisieren. Dies führte zu der Überlegung, wie von den analysierten und visualisierten Testergebnissen hinsichtlich der Fehlerlokalisierung Gebrauch gemacht werden kann. Es wurde illustriert, wie mit Hilfe der modellgetriebenen Visual ScatterUnit die formalen Modelle wiederholt schrittweise verfeinert werden können, um zutreffende Hypothesen bezüglich der Versagensursache treffen zu können. Auf diese Weise zeigte sich, dass sich der modellgetriebene Ansatz kombiniert mit einer entsprechenden Visualisierung ausgezeichnet einsetzen lässt, um diesen kognitiven iterativen Debuggingprozess effizient und robust auszuführen.

4. Technische Realisierung des Testrahmenwerks

Wie im letzten Kapitel ersichtlich wurde, bedingt ein optimierter Einsatz des Tests eine robuste Zusammenarbeit seiner verschiedenen Komponenten entlang der Architekturhierarchie, angefangen von der auf den Sensorknoten laufenden Testinfrastruktur ScatterUnit bis hin zur Domänensprache, die auf Basis des dazu gehörigen Metamodells definiert ist. Daher ist es an dieser Stelle angebracht, einen genaueren Blick auf die im Hintergrund laufenden Prozesse zu werfen, auf denen die kohärente, zielgerichtete Gesamtfunktionalität des Rahmenwerks beruht. Dabei folgt die Behandlung der Komponenten der Werkzeugkette einer *top-down* Betrachtung. Zunächst werden die Eigenschaften der Domänensprache (DSL) beschrieben [vgl. Kame 08], dann wird erörtert, wie anhand des Metamodells die Modellierungssprache-eigenschaften in Visual ScatterUnit entworfen und realisiert wurden. Dies führt zur Validierung [vgl. Kame 08] der DSL, welche auf den Eigenschaften des Metamodells fußt, und der Frage, inwiefern die Validierung den darauf folgenden Codegenerierungsprozess [vgl. Kame 08] steuert. Die wichtigsten Techniken der Generierung von Code-Artefakten zur Testautomatisierung in Visual ScatterUnit werden diskutiert. Abschließend werden die Dienste [vgl. Kame 07] der Testautomatisierungsschicht ScatterUnit dargestellt und geschildert, wie diese den verteilten Test auf Sensorebene bewerkstelligen.

4.1. Entwurf und Realisierung der Domänensprache (DSL)

Bei der Betrachtung des Modells für den Testfall des Routingprotokolls im vorangegangenen Kapitel wurde ersichtlich, dass es sich bei der Domänensprache (DSL) zur Modellierung eines Testfalls um zwei Arten von Diagrammen handelt: Das Aktivitätsübersichtsdiagramm (vgl. Abbildung 20 im letzten Kapitel) zur Koordination des Testverlaufs einerseits und die Detaillierungsdiagramme der einzelnen Ablaufstränge (Aktivitäten) andererseits, aus denen sich das Aktivitätsdiagramm zusammensetzt. Der Ausgangspunkt für die Modellierung des Tests ist das Aktivitätsdiagramm, anhand dessen der Testverlauf spezifiziert wird. Hierfür wird durch die Ablaufstränge im Aktivitätsdiagramm die Reihenfolge und die Gliederung der ablaufenden Ereignisse während der verteilten Ausführung des Testszenarios auf den verschiedenen Sensorknoten festgelegt. Zudem wird der Aspekt der Parallelität ausschließlich auf dieser Modellierungsebene in Betracht gezogen. Dazu muss der Softwaretester (Modellierer) eine für seinen Testverlaufsentwurf geeignete Gliederung der Ablaufstränge zwischen den Synchronisationsbalken entlang des Aktivitätsdiagramms festlegen. Sowohl die Semantik als auch die Syntax des Aktivitätsdiagramms müssen diesen Anforderungen genügen, was auf Metamodellierungsebene (dem Rückrat jeglicher Domänensprache) zu bewerkstelligen ist.

Es muss daher an dieser Stelle genauer unter die Lupe genommen werden, wie das Aktivitätsdiagramm zum Zweck der Koordination des verteilten Testverlaufs auf Metamodellierungsebene konzipiert wird. Abbildung 28 zeigt den für den Aufbau des Aktivitätsdiagramms relevanten Ausschnitt des Metamodells der Domänensprache. Dem Ausschnitt des Metamodells entsprechend benötigt jedes Aktivitätsübersichtsdiagramm einen Startzustand zur Generierung des Initialisierungscodes des Tests. Die Entität Startknoten besitzt jedoch die Kardinalität 0..1 anstelle von 1, was bedeutet, dass das Vorhandensein eines Startknotens im Aktivitätsdiagramm nicht zwingend ist – was offensichtlich semantisch nicht korrekt ist.

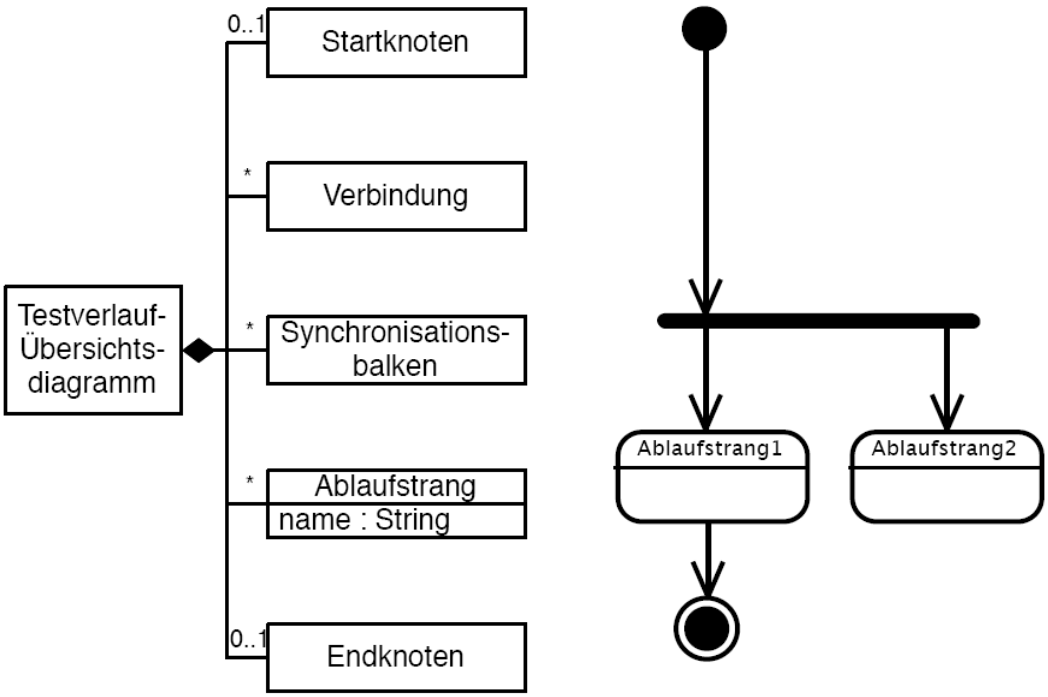


Abbildung 28 Der relevante Ausschnitt des Metamodells (links), das die Entitäten des Aktivitätsdiagramms (rechts) sowie deren Zusammenhänge veranschaulicht [Kame08]

Dahinter verbirgt sich der Umstand, dass editorbedingt ein mit Hilfe des generativen Editors neu zu erstellendes Diagramm anfangs keine Diagrammkonstrukte beinhalten kann. Da nun das Metamodell als Grundlage für die automatisierte Erstellung des generativen Editors fungiert, muss man dieser technischen Herausforderung bereits auf Metamodellebene begegnen. Zwar wird dies syntaktisch zugelassen, sobald man jedoch das erste Modellkonstrukt des Aktivitätsdiagramms gebildet hat, muss unmittelbar danach auf die in diesem Zusammenhang getroffene semantische Regel verwiesen werden.

Hier kommt die Livevalidierung des Modells zum Zuge. Zu diesem Zweck müssen die entsprechenden Randbedingungen mittels der Sprache Checks (vgl. Kapitel 2.5) definiert werden. Im Programmauszug 2 werden die dazu notwendigen Checks-Ausdrücke[39] aufgelistet. Ein wichtiger semantischer Aspekt des Aktivitätsübersichtsdiagramms ist die Struktur der Assoziation (Verbindung) zwischen den Diagrammelementen. Abbildung 29 zeigt den relevanten Ausschnitt des Metamodells. Analog zur editorbedingten Besonderheit lässt das Metamodell beliebig viele ein- und ausgehende Assoziationen für jede Diagrammentität zu. Jedoch dürfen zum Startknoten keine Assoziationen eingehen und aus dem Endknoten keine ausgehen. Diese Modellierungsvorgabe muss auf Editorebene eingehalten werden. Um dies zu gewährleisten, werden entsprechende OCL-Randbedingungen an die Metamodellkonstrukte gebunden (vgl. Programmauszug 3). Dies bewirkt, dass jegliche falsche Assoziation bei den betroffenen Diagrammentitäten von vornherein im Editor ausgeschlossen wird.

[39] Eine Auflistung der in Visual ScatterUnit definierten Randbedingungen findet sich im Anhang 11.5.1.

```
context testcase::Testverlauf–Uebersichtsdiagramm
ERROR "Es muss genau ein Startknoten existieren." :
this.startknoten != null;
context testcase::Testverlauf–Uebersichtsdiagramm
ERROR "Es muss genau ein Endknoten existieren." :
this.endknoten != null;
context testcase::Testverlauf–Uebersichtsdiagramm
ERROR "Es muss mindestens ein Ablaufstrang existieren." :
this.ablaufstraenge.size > 0;
```

Programmauszug 2 Ausdrücke der Sprache Checks zur Überprüfung der semantischen Regel des Vorhandenseins eines Start- und Endknotens im Aktivitätsdiagramm zur Modellierungszeit [Kame08]

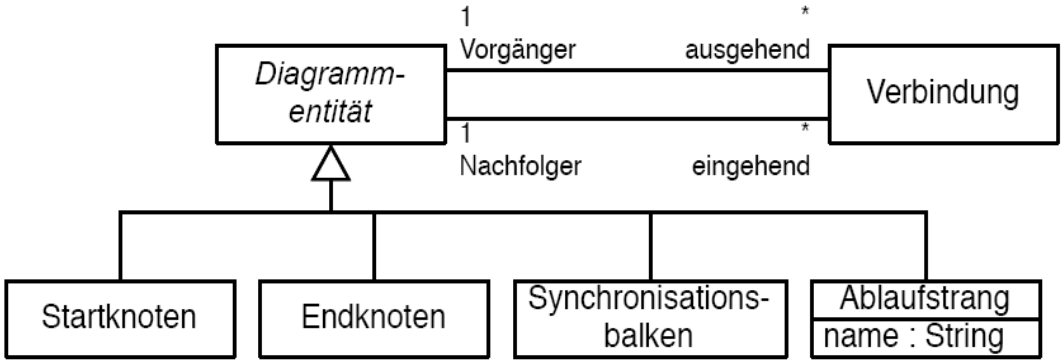


Abbildung 29 Assoziationen zwischen den Diagrammentitäten (Ausschnitt aus dem Metamodell, UML-Notation) [Kame08]

```
not self.oclIsTypeOf(Startknoten)
not self.oclIsTypeOf(Endknoten)
```

Programmauszug 3 OCL-Ausdruck zum Start- und Endknoten des Aktivitätsdiagramms [Kame08]

Hier wurden die semantischen Regeln nicht durch die Livevalidierung anhand von Checks-Ausdrücken definiert (was auch möglich wäre), sondern bereits auf Metamodellebene mit Hilfe von an die Metamodellelemente gebundenen OCL-Ausdrücken angegeben. Bei den semantischen Regeln handelt es um solche, die sich stark an der Notation der Modellentitäten (wie z.B. deren Verknüpfung im Editor) orientieren und weniger an modellierten Aspekten der zugrunde liegenden Domäne. Daher werden solche Randbedingungen wie oben beschrieben mit Hilfe von OCL-Ausdrücken formuliert, um deren Modellierbarkeit im Editor auszuschließen. Dies führt dazu, dass die betroffenen Modellentitäten im Editor mit der Maus nicht verknüpft werden können (vgl. Abbildung 30).

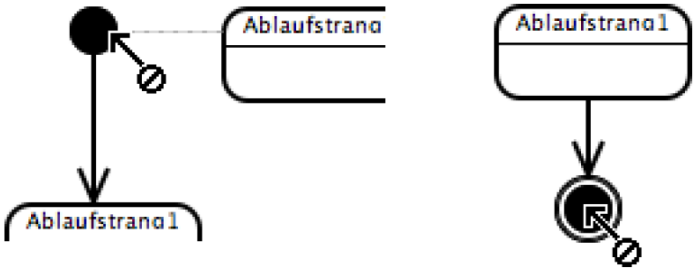


Abbildung 30 Verbindungen dürfen bei keinem Startknoten enden (links). Verbindungen dürfen ihren Ausgangspunkt nicht in einem Endzustand haben (rechts). [Kame08]

Ferner muss gewährleistet werden, dass beispielsweise im Fall des Startknotens genau eine ausgehende Assoziation definiert wird. Obwohl dies analog zum letzten Beispiel mit Hilfe von OCL-Ausdrücken bewerkstelligt werden könnte, wurden die semantischen Regeln dazu mittels der Livevalidierung spezifiziert, da diese eine visuelle Fehlermeldung im Editor geben kann (vgl. Programmauszug 4):

```
context testcase::Startknoten
ERROR "Der Startknoten muss genau eine ausgehende Verbindung haben." :
this.ausgehend.size == 1;
```

Programmauszug 4 OCL-Ausdruck zum Start- und Endknoten des Aktivitätsdiagramms [Kame08]

Die zweite Diagrammart der Domänensprache modelliert die Detaillierung eines Ablaufstrangs, in dem zusätzlich die während des Tests entstandenen Ergebnisse eines Sensorknotens abschließend visualisiert werden. Da jeder Ablaufstrang eine Aktivität des Testfalls repräsentiert, werden durch das Detaillierungsdiagramm die technischen Aspekte modelliert, die zur Realisierung der jeweiligen Funktionalität des Ablaufstrangs bzw. der jeweiligen Aktivität der Testdurchführung notwendig sind. Solche im Detaillierungsdiagramm modellierten technischen Belange fungieren dann als Eingabeparameter für den Testgenerator, um daraus den zugehörigen Infrastrukturcode[40] automatisiert zu generieren. Das Ablaufstrangdiagramm liefert somit einen tieferen Einblick ins Innere eines bestimmten Sensorknotens hinsichtlich des Verhaltens der darauf laufenden (zu testenden) Anwendungen. Zudem wird die Reihenfolge der Ausführung der einzelnen Aktionen als lineare Sequenz festgelegt, die auf dem jeweiligen Knoten läuft. Der Modellierer muss festlegen, welche Aktionen zu einem Ablaufstrang zusammengestellt werden, er muss jedoch nicht für die Parallelität der Ausführung Sorge tragen, da diese, wie bereits erwähnt, ausschließlich durch die Synchronisationsbalken des Aktivitätsübersichtsdiagramm koordiniert wird. Abbildung 31 zeigt den relevanten Ausschnitt des Metamodells zur Definition des detaillierten Diagramms. In diesem Ausschnitt finden sich die im letzten Kapitel bezüglich des Routingprotokolls behandelten Modellentitäten wieder:

[40] Beim Infrastrukturcode beim Einsatz Visual ScatterUnit handelt es sich beispielsweise um den Code zur Erstellung und Bearbeitung der Log-Einträge, zur Analyse der Testergebnisse, Dienste von ScatterUnit, Steuerung und Koordination des Testverlaufs und Visualisierung der Testergebnisse im zugehörigen Modell.

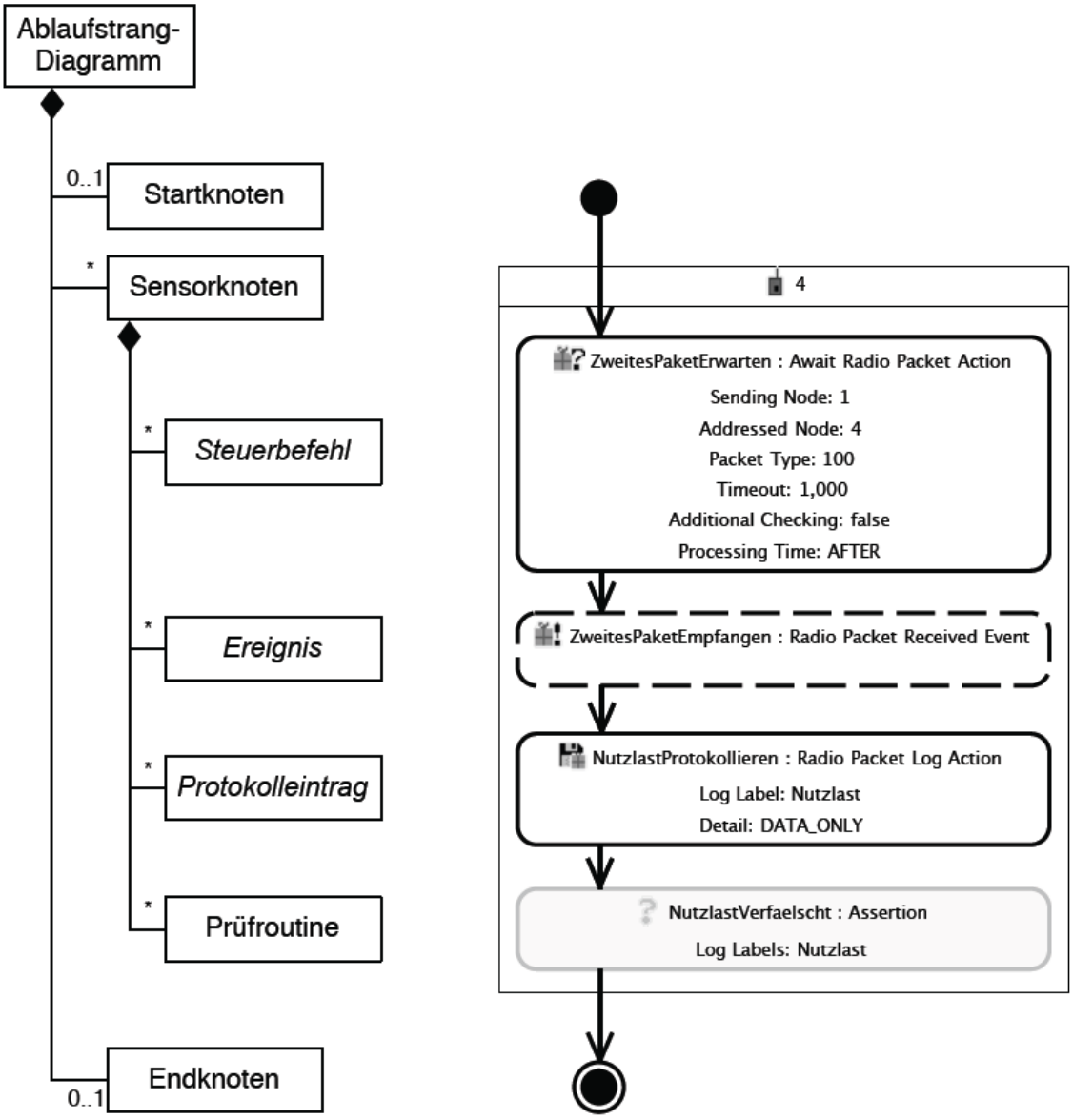


Abbildung 31 Der relevante Ausschnitt des Metamodells in UML-Notation (links) der Entitäten des Detaillierungsdiagramms (rechts) [Kame08]

1. Steuerbefehle, die bestimmte Aktionen des Testfalls bewirken. Es handelt sich dabei um Aufrufe der zu testenden Anwendung zur Durchführung des Testfalls, wie z.B. das Versenden eines Pakets mittels der dazugehörigen Methode des Routingprotokolls (Testobjekt), bzw. Aufrufe von Diensten der Testschicht ScatterUnit, wie der Dienst zur Topologieänderung, um den Sendebereich zu simulieren. Dabei werden Pakete beim Empfang einfach gefiltert.[41]

[41] Bei den gefilterten Paketen handelt es sich um diejenigen, die aus den Sensorknoten stammen, die außerhalb des Sendebereichs liegen sollen.

2. Ereignisse, die als Benachrichtigung auf die korrekte Ausführung eines erwarteten Steuerbefehls bzw. einer erwarteten Aktion fungieren, wie z.B. der erfolgreiche Empfang eines erwarteten Paketes.
3. Protokolleinträge dienen als Log-Mechanismus zur Protokollierung relevanter Informationen des Testverlaufs. Zwar werden alle Steuerbefehle und Ereignisse standardmäßig mit Hilfe entsprechender Dienste von ScatterUnit protokolliert, jedoch handelt es sich hier um zusätzliche, anwendungsspezifische Informationen, die entsprechend der zur Fehlersuche aufgestellten Hypothesen angegeben werden.
4. Prüfroutinen sind auf dem Rechner laufende Module, welche die von den Protokolleinträgen eingehenden Informationen auf deren Korrektheit prüfen. Somit fungieren die Prüfroutinen als Prüfmechanismen, anhand derer die zur Fehlersuche aufgestellten Hypothesen verifiziert werden. Das Ergebnis wird dann in entsprechenden Modellkonstrukten visualisiert. Für jeden Typ eines Protokolleintrags ist eine bestimmte Prüfroutine zuständig. Daher muss die Domänensprache (DSL) die eindeutige Zuordnung der einzelnen Protokolleinträge und ihrer Prüfroutinen auf Modellebene gewährleisten. Dies geschieht durch eindeutige Modellmarkierungen in den jeweiligen Modellkonstrukten, wie in Abbildung 32 skizzenhaft veranschaulicht wird.

Da der überwiegende Codeanteil der Modellentitäten automatisiert generiert wird, müssen die Modellentitäten über bestimmte Parameter verfügen, die als Eingabe für den Codegenerator fungieren, um ihren Code auf Basis der korrespondierenden Templates vollständig automatisiert zu generieren. Abbildung 33 zeigt eine Ergänzung des Metamodellausschnitts aus Abbildung 31, bei dem die Modellentitäten zwecks automatisierter Codegenerierung parametrisiert werden. Die Attribute verweisen daher im Allgemeinen auf die häufig benutzten Dienste der Testschicht ScatterUnit. Werden nun die Dienste von ScatterUnit ausgebaut, so kann der relevante Teil des Metamodells problemlos entsprechend erweitert werden. Die Funktionalität ScatterUnits kann infolgedessen auf flexible Weise um weitere Dienste ergänzt werden. Dies trägt der Dynamik der WSN-Domäne und der damit verbundenen Vielfalt von zu testenden Anwendungen Rechnung.

Demzufolge wurde das Metamodell, wie das in UML-Notation erstellte Diagramm (vgl. Abbildung 90 im Anhang 11.4) veranschaulicht, auf Basis von hierarchischer Generalisierung bzw. Vererbung zwischen den Metamodellentitäten derart strukturiert, dass das Einflechten beliebiger Steuerbefehle zu einem späteren Zeitpunkt flexibel möglich ist. Werden gewisse Steuerbefehle wiederholt zum Testen unterschiedlicher Anwendung in Anspruch genommen, so ist es für solche Befehle effizient, entsprechende Metamodellinstanzen zu definieren, die die adäquate Angabe der für den Codegenerator erforderlichen Parameter im Testfallmodell ermöglichen.

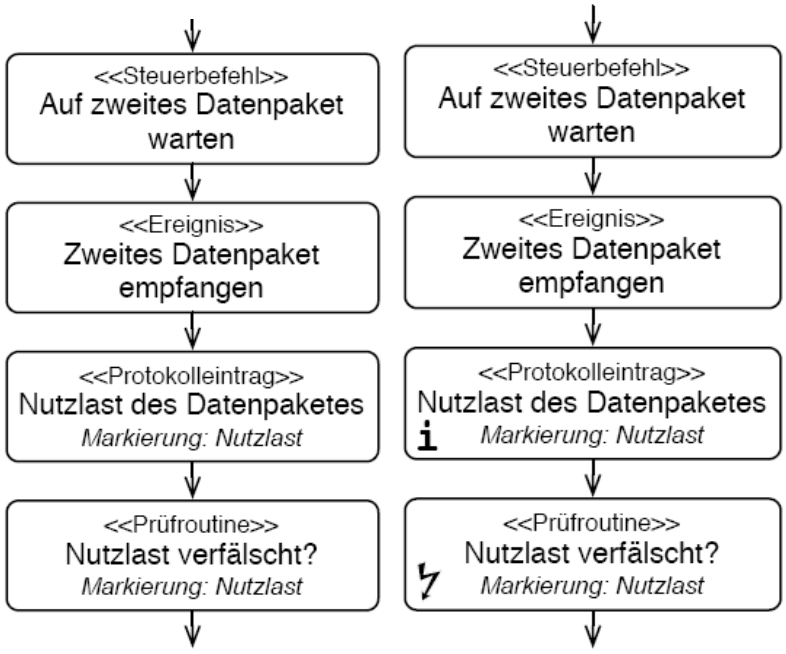


Abbildung 32 Die Markierungen (hier Nutzlast) fungieren als Bindungsmechanismus zwischen den Protokolleinträgen und deren Prüfroutinen. Die Meldungen der Überprüfungen werden anschließend im Modell angezeigt (rechts). [Kame08]

Ein Beispiel hierzu sind die Modellentitäten „Warten auf Paket“ und „Ändern der Topologie“ aus Abbildung 33, bei denen es sich um oft zu verwendende Funktionalitäten in der WSN-Domäne handelt. Diese werden daher als Dienste ScatterUnits angeboten, wobei sie als fixer Bestandteil der Domänensprache unterstützt werden, was bedeutet, dass für solche Dienste Metamodellinstanzen definiert werden. Die Modellkonstrukte solcher Dienste stehen somit dem Nutzer in der Palette des generativen Editors zur Verfügung (vgl. Abbildung 34), da der generative Editor, wie in Kapitel 4 erwähnt, aus dem Metamodell auf einfache Weise erneut automatisiert generiert werden kann. Damit passt sich die Struktur des Editors den Änderungen der Anforderungen an, die entstehen, wenn unterschiedliche Anwendungen getestet werden. Ergeben sich neue Anforderungen in der Testdomäne, so werden die entsprechenden Dienste von ScatterUnit einmalig implementiert und ins Metamodell eingebettet. Daraus wird der Editor automatisiert erneut erstellt, und die neuen Dienste stehen dem Nutzer in Form von visuellen Modellen in der Palette des Editors zur Verfügung. Die Offenheit der Architektur der Testinfrastruktur steigert sich dadurch deutlich.

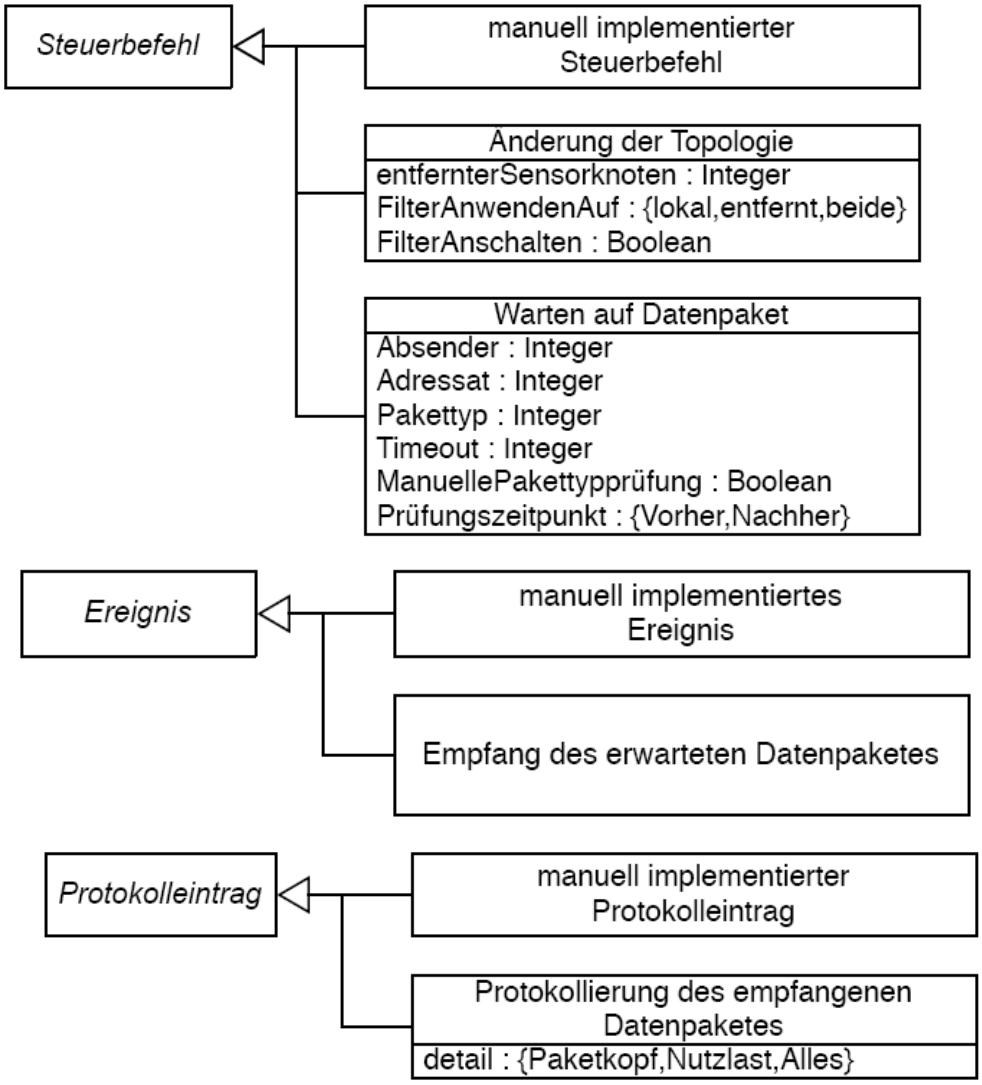


Abbildung 33 Ausschnitt des attributierten Metamodells des Detaillierungsdiagramms [Kame08]

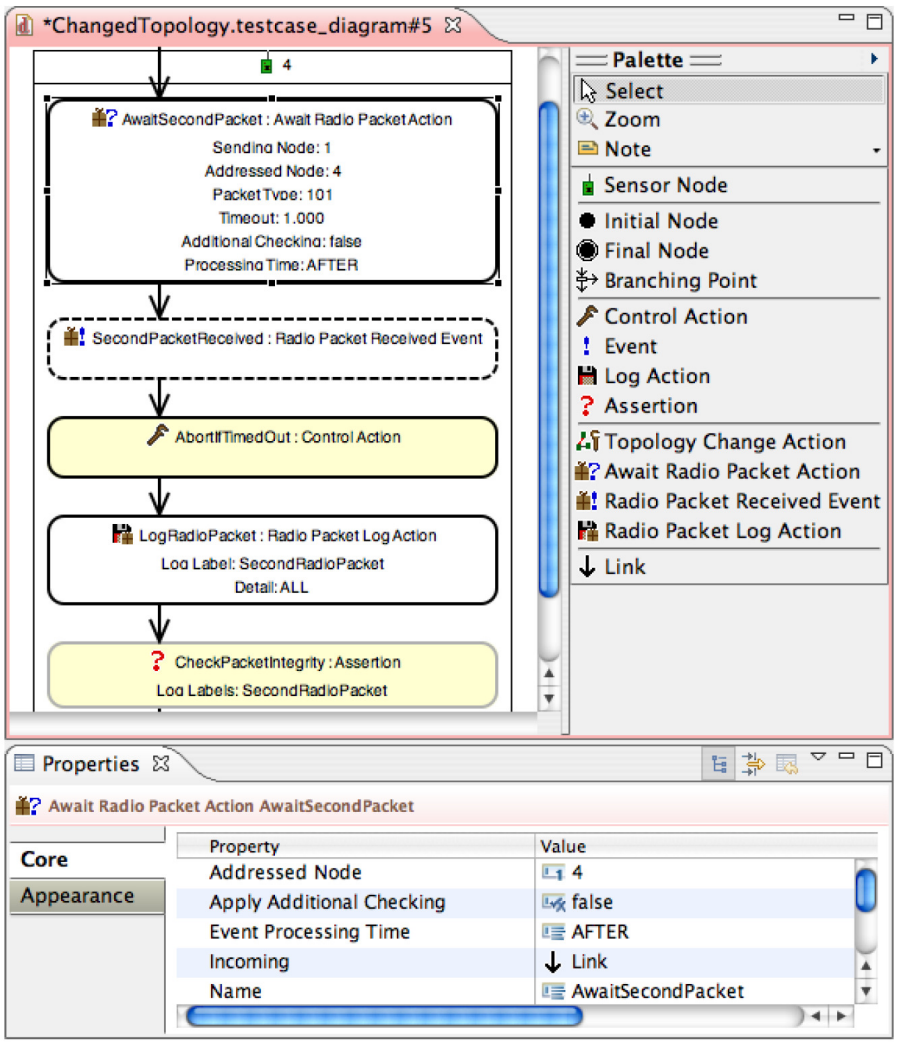


Abbildung 34 Der generative Editor des Rahmenwerks [Kame08]

4.2. Live-Validierung der Modelle

Wie bereits in 2.5 erläutert, fußt die Realisierung der Live-Validierung auf der Zusammenarbeit zwischen dem openArchitectureWare Generator-Rahmenwerk und dem generativen GMF-Editor. Die Bindung wird durch den von openArchitectureWare standardmäßig gelieferten GMF-Adapter hergestellt, durch den jegliche im Editor stattgefundene Modelländerung überwacht und gleichzeitig gegen die in Checks definierten Ausdrücke validiert wird (vgl. Abbildung 10 in 2.5). Die Verwendung der Live-Validierung wurde im letzten Abschnitt im Kontext der Modellierung des Testverlaufs mittels des Aktivitätsdiagramms erörtert. Dabei wurde gezeigt, wie durch Checks-Ausdrücke ans Modell gebundene semantische Randbedingungen zur robusten Steuerung des Testverlaufs beitragen. In diesem Abschnitt soll der Fokus breiter auf die Wirkung der Live-Validierung gerichtet werden. Es sind 136 unter-

schiedliche Checks-Ausdrücke[42] in Visual ScatterUnit definiert worden, die gemeinsam eine umfassende Beherrschung der Charakteristika der unterliegenden Domäne auf hoher abstrakter Ebene – nämlich der DSL – ermöglichen. Da selbstverständlich nicht auf jeden einzelnen Ausdruck eingegangen werden kann, werden im Folgenden prägnante Randbedingungen sowie deren Effekte diskutiert.

So wird beispielsweise durch die Definition des folgenden Checks-Ausdrucks gewährleistet, dass Bezeichner von Aktivitäten im Modell keine Sonderzeichen enthalten (vgl. Abbildung 35 und Programmauszug 5):

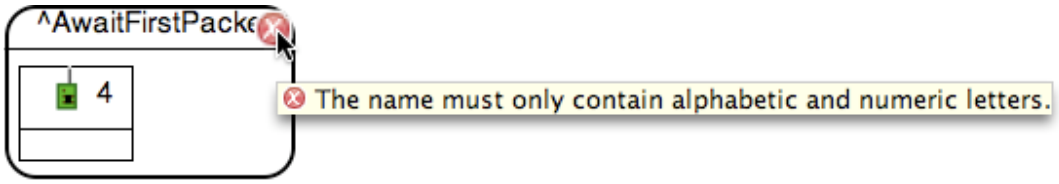


Abbildung 35 Live-Validierung deckt den ungültigen Namen einer Aktivität auf

```
context testcase::ControlPath
ERROR "The name must only contain alphabetic and numeric letters." :
this.name.matches("[\\p{Alpha}\\p{Digit}]*");
```

Programmauszug 5 Checks-Ausdruck zur syntaktischen Prüfung [Kame08]

Eine solche Einschränkung ist unerlässlich, da die Namen der Aktivitäten des Modells in den Methodennamen im generierten Code vorkommen. Würde dies zur Modellierungszeit der Live-Validierung nicht überprüft, ließen sich die betroffenen Methoden nicht kompilieren. Hier sieht man erneut, wie die Modell-Validierung im Kontext des modellgetriebenen Paradigmas wesentlich zur Steigerung der Robustheit des Entwicklungsprozesses beiträgt. Das hier angeführte Beispiel zeigt, dass Fehler in früheren Phasen des Entwicklungsprozesses ausgeschlossen werden können – nämlich auf Modellierungsebene – und nicht erst auf der fehlerflüchtigen Codierungsebene. Je früher ein Fehler im Entwicklungsprozess aufgedeckt wird, desto unerheblicher sind seine Auswirkungen. Zwar wird eine Vielzahl syntaktischer Randbedingungen auf Modellebene durch die Live-Validierung überprüft, die Live-Validierung kommt jedoch vor allem zum Tragen, wenn es sich um die Validierung semantischer Randbedingungen handelt. Hier entfaltet sich der wahre Nutzen der Modellüberprüfung in Echtzeit. So ist es z.B. unerlässlich, Zyklen im Aktivitätsdiagramm auszuschließen, da durch den Baum des Aktivitätsdiagramms der Fluss des Testverlaufs bezüglich des zeitlichen Voranschreitens der auf den Knoten verteilten Tests modelliert wird (vgl. Abbildung 36).

Daher setzt sich der Testfall aus einer linearen Sequenz von Aktivitäten zusammen, die mit einem Startknoten beginnt und einem Endknoten endet, wodurch die Ausführung des Testfalls sowie die dazu nötige Codegenerierung gesteuert wird (vgl. Programmauszug 6).

[42] Siehe Anhang 11.5.

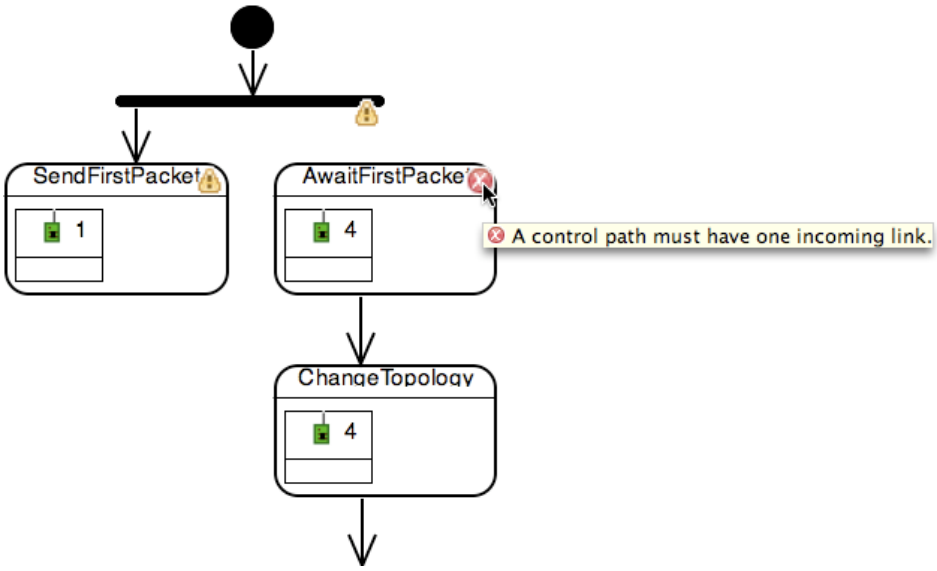


Abbildung 36 Live-Validierung deckt auf, dass eine Aktivität keinen Vorgänger besitzt

```
context testcase::ControlPath
ERROR "A control path must have one incoming link." :
this.incoming.size == 1;
context testcase::ControlPath
ERROR "A control path must have at maximum one outgoing link." :
this.outgoing.size <= 1;
context testcase::TestCaseDiagram
ERROR "Exactly one initial node must exist." :
this.initialNode != null;
context testcase::InitialNodeOfTheTestCase
ERROR "The initial node must not have any incoming links." :
this.incoming.size == 0;
context testcase::TestCaseDiagram
ERROR "Exactly one final node must exist." :
this.finalNode != null;
context testcase::FinalNodeOfTheTestCase
ERROR "The final node must not have any outgoing links." :
this.outgoing.size == 0;
```

Programmauszug 6 Checks-Ausdruck zu Abbildung 36 [Kame08]

Des Weiteren werden parallel laufende Abläufe während des Testfalls ermöglicht. Dabei muss jedoch beachtet werden, dass jeder individuelle Sensorknoten aus technischen Gründen keine parallele Abarbeitung der auf ihm laufenden Prozesse unterstützt. Daher können verschiedene, für den Test nötige simultane Aktivitäten nur auf unterschiedlichen Sensorknoten verteilt ausgeführt werden. Demgemäß muss die Live-Validierung gewährleisten, dass parallel laufende Aktivitäten[43] im Aktivitätsdiagramm unterschiedlichen Sensorknoten zugewiesen sind (vgl. Abbildung 37 und Programmauszug 7).

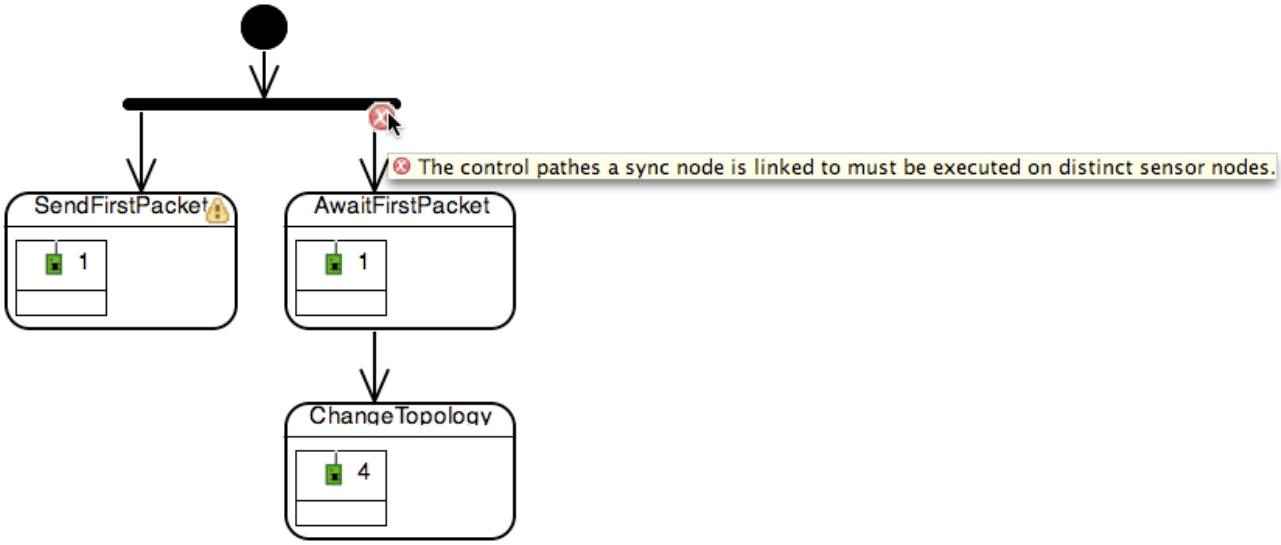


Abbildung 37 Live-Validierung entdeckt eine unzulässige parallele Ausführung

```
context testcase::SyncNode

ERROR "The control pathes a sync node is linked to must be executed on distinct
sensor nodes." :

((List[testcase::ControlPath])this.outgoing.target.select(e | e.metaType

== testcase::ControlPath)).getDeploymentIdForFirstAction().size ==

((List[testcase::ControlPath])this.outgoing.target.select(e | e.metaType

== testcase::ControlPath)).getDeploymentIdForFirstAction().toSet().size;
```

Programmauszug 7 Checks-Ausdruck zu Abbildung 37 [Kame08]

Der durch die Live-Validierung bereits auf Modellierungsebene bewirkte Ausschluss des Vorkommens von Zyklen im Testverlaufsmodell führt zu einer Reduzierung der Komplexität des darauf folgenden Codegenerierungsprozesses. Dies wird dadurch erreicht, dass während der Einbindung der notwendigen Templates nicht laufend überprüft werden muss, ob die für die betroffenen Modellentitäten geltenden Randbedingungen verletzt wurden oder nicht. So kann die Einbindung der nötigen Temp-

[43] Das sind diejenigen Aktivitäten, die im Diagramm zwischen zwei aufeinander folgenden Synchronisationsbalken modelliert werden.

lates unmittelbar an den Anstoß des Generierungsprozesses anschließen, ohne dass die Gefahr besteht, durch im Modell vorhandene Zyklen in eine endlose Schleife zu geraten.

Neben der Reduzierung der Komplexität des Codegenerierungsprozesses leistet die Live-Validierung einen wesentlichen Beitrag zur Steigerung der Qualität des für den Testfall benötigten Codes. Betrachtet man beispielsweise die benutzerdefinierten Protokolleinträge (Log-Aktionen) zur Fehlersuche im Detaillierungsdiagramm, so kann durch die Live-Validierung garantiert werden, dass an jeden solchen Protokolleintrag die korrespondierende Prüfroutine durch das entsprechende Markierungslabel gebunden wird. Außerdem überprüft sie, dass die im Modell verwendeten Markierungslabel über unterschiedliche Bezeichner verfügen.

Infolgedessen erwartet jede Prüfroutine während der Analyse der Testergebnisse nach Ausführung des Tests ihre zugehörigen Protokolleinträge. Erhält sie diese nicht, kann angenommen werden, dass der Testverlauf oder die Sammlung der Testresultate nicht ordnungsgemäß stattgefunden hat. Entsprechend kann eine im Generat für diesen Fall maßgeschneiderte Fehlermeldung automatisiert generiert werden. Durch die Garantie, dass lediglich diejenigen Logaktionen im Test ausgeführt werden, die tatsächlich im formalen Modell vorkommen, wird die Robustheit des Testverlaufs erhöht. Andernfalls deckt die Live-Validierung den entsprechenden Fehler auf, wodurch der dazugehörige Code automatisiert gemäß dem Testverlauf generiert wird, was dessen Qualität merklich erhöht.

4.3. Die Codegenerierung

Nach Beendigung der Modell-Validierung ist gewährleistet, dass die formalen Modelle weder die syntaktischen noch die semantischen Randbedingungen der unterliegenden Domäne verletzen. Erst danach wird der Codegenerierungsprozess angestoßen, indem die für das Modell benötigten Templates eingebunden werden. Wie in 2.5 beschrieben, beinhalten die Templates des oAW-Rahmenwerks die notwendigen Anweisungen um die Codegenerierung zu steuern. Dies bewirkt, dass die zum jeweiligen Modell gehörenden Codeartefakte aus den Templates emmitiert und zusammengefügt werden. So erzeugt beispielsweise das folgende Template (vgl. Programmauszug 8) die Initialisierungsroutine zur Topologieänderung des WSNs (vgl. 3.3) um den Testfall des DSDV-Routingprotokolls anzustoßen.

```
<< DEFINE mainFile FOR Integer >>
   << FILE "main.c" >>
      #include "main.h"
      bool Test SetUp() {
        << IF changingTopology() >>
           PosSim on();
            << ENDIF >>
           return True;
      }
   << ENDFILE >>
<< ENDDEFINE >>
```

Programmauszug 8 Template der Initialisierungsroutine zur Topologieänderung des WSNs [Kame08]

Die für die Topologieänderung zuständige SetUp() Methode wird in die Zieldatei main.c geschrieben. Die Methode liefert einen wahren Wert und veranlasst die Topologieänderung im Fall, dass die changingTopology() Anweisung ebenfalls als wahr ausgewertet wird. Dies führt dazu, dass die Anweisung PosSim_on() aufgerufen wird, die die Aktivierung des Simulationsdiensts der ScatterUnit veranlasst. Es stellt sich nun die Frage, warum vor dem Aufruf von PosSim_on() die Anweisung changingTopology() aufgerufen wird (vgl. Programmauszug 9). Bei changingTopology() handelt es sich um eine Xtend-Anweisung (eine Erweiterung), die vom oAW Codegenerator unterstützt wird. (Das Konzept wurde bereits in 2.5 erläutert.) Hier findet sich ein anschauliches Anwendungsbeispiel zum Einsatz dieser Erweiterungen, wodurch die Codegenerierung effizienter vonstatten geht: Der Auszug zeigt die Implementierung der changingTopology-Erweiterung. Dadurch wird im zugrunde liegenden formalen Modell überprüft, ob sie die Entität vom Typ TopologyChangeAction[44] enthält, wodurch der Code zur Inanspruchnahme des Simulationsdienstes generiert wird. Ist dies der Fall, wird PosSim_on() aufgerufen, um den Dienst zu aktivieren. Somit wird deutlich, welchen Beitrag Erweiterungen hinsichtlich der Modellanalyse zum Zweck der Steuerung der Codegenerierungsdirektiven leisten.

```
Boolean changingTopology() :
  model().controlPathes.sensorNodes.controlActions
    .exists(e | e.metaType == TopologyChangeAction);
```

Programmauszug 9 Definition von geschützten Bereichen innerhalb des Templates [Kame08]

Insgesamt wird also mittels Xtends die Xpand-Sprache auf eine modulare Weise operativ erweitert, so dass die Funktionalitäten von Xtends separat und ohne einen Eingriff in den strukturellen Aufbau der Xpand-Sprache und damit den Templates eingesetzt werden können. Dies erleichtert die Programmierung solcher Templates erheblich.

Ein weiteres wichtiges Merkmal der Xpand-Sprache sind die so genannten *geschützten Bereiche* (protected regions). Dies sind im Generat ausgewiesene Bereiche, die beim erneuten Generierungsdurchlauf nicht überschrieben werden. Der Block der geschützten Bereiche wird vor der erneuten Codegenerierung durch das Generierungs-Rahmenwerk gepuffert und erst danach wieder an seiner vorgesehenen Stelle in den neu generierten Code integriert. Daher muss jedes Segment eines geschützten Bereichs eindeutig identifizierbar sein. Dies wird durch die Markierung der Segmente mittels *Universally Unique Identifier* (UUID) erreicht.[45] Solche geschützten Bereiche sowie deren Inhalt werden vom Benutzer manuell festgelegt. Daher werden sie insbesondere dafür verwendet, den manuellen Codeanteil vom automatisierten auf flexible Art deutlich zu trennen.

---

[44] Siehe dazu das Diagramm des Metamodells von Visual ScatterUnit (vgl. Abbildung 90 im Anhang 11.4), in dem der Typ der Entität definiert wird.

[45] „Ein UUID besteht aus einer 16-Byte-Zahl, die in fünf Gruppen unterteilt wird. In ihrer Normalform sieht eine UUID so aus: 550e8400-e29b-11d4-a716-446655440000 [...]. Daher können UUIDs beliebig ohne zentrales Kontrollorgan erzeugt und zur Kennzeichnung eingesetzt werden, ohne Gefahr zu laufen, dass eine gleiche UUID für etwas anderes verwendet wird.“ [WebUUID]

Es gibt, wie im letzten Kapitel geschildert, verschiedene Szenarien in Visual ScatterUnit, bei denen der Code nicht vollständig automatisiert generiert werden sollte. Gerade für Testzwecke müssen bestimmte Aspekte bezüglich der zu testenden Anwendungen spezifisch vom Benutzer eingespeist werden. Beispielsweise werden Codeanteile bestimmter Aktionen bzw. Ereignisse der Testfälle oft vom Benutzer manuell implementiert, da diese sich stark an der zu testenden Anwendung orientieren. Dies wird durch den folgenden Programmauszug 10 aus Visual ScatterUnit zur Generierung eines Testfalls für das DSDV Routing-Protokoll demonstriert.

Der Programmauszug zeigt ein vollständiges Codeartefakt, bestehend sowohl aus manuellem, als auch aus generiertem Code. Wie der Programmauszug in den Zeilen 10-13 erkennen lässt, findet eine Interaktion zwischen dem manuellen und dem automatisiert generierten Code statt. Diese Zusammenarbeit wird möglich durch die in Zeile 2 definierte Boolesche Variable. Wie man sieht, müssen hier und da gegebenenfalls kleine Eingriffe vom Entwickler vorgenommen werden bis das Endprodukt des generierten Softwareartefakts vorliegt. Programmauszug 11 zeigt das zum obigen Code gehörende Template. Der geschützte Bereich im Template wird durch die Tags PROTECT und ENDPROTECT definiert. Die TODO Kommentare, die sich innerhalb dieses Bereichs befinden, werden manuell nach der Codegenerierung anwendungsspezifisch vervollständigt.

```
1 // ControlAction InitNode1
2 bool bAbortTestCaseExecution = False;
3
4 /* PROTECTED REGION ID(MovingReceiverTestCase Init InitNode1) ENABLED START */
5
6 DSDV init();
7
8 /* PROTECTED REGION END */
9
10 if (bAbortTestCaseExecution) {
11 Test abort(0);
12 return;
13 }
```

Programmauszug 10 Einsatz von geschützten Bereichen für den manuellen Code [Kame08]

```
<< PROTECT CSTART '/*' CEND '*/' ID this.protectedRegionId(testCaseName) >>
// TODO: Implement ControlAction this.name here.
// TODO: Set bAbortTestCaseExecution to True if a failure occured.
<< ENDPROTECT >>
```

Programmauszug 11 Definition von geschützten Bereichen innerhalb des Templates [Kame08]

4.4. Die Dienste der Testschicht ScatterUnit

ScatterUnit [Kame07] stellt die notwendigen Dienste zur Verfügung, die die Automatisierung des Testablaufs von ScatterWeb-Anwendungen auf Sensorknotenebene ermöglichen. Hierfür ist es unerlässlich, die Funktionalitäten der Testschicht sowohl direkt auf den Sensorknoten als auch auf der Basisstation (PC) laufen zu lassen. Zu diesem Zweck wurde einerseits die Laufzeitumgebung der Sensorknoten mit Hilfe der Bibliothek *libScatterUnit* bestückt. Anderseits wurde ein Rahmenwerk namens *ScatterUnit Testautomation* realisiert. Dieses Java-basierte Rahmenwerk läuft auf dem Rechner der Basisstation, an den die Senke angeschlossen ist und sorgt für die automatische Initialisierung bzw. Konfiguration notwendiger Softwaremodule der Sensorknoten um den Testablauf anzustoßen. Wurde die Ausführung des Testfalls gestartet, schreitet der Prozess wie folgt voran:[46]

Jeder am Testszenario beteiligte Sensorknoten empfängt zum Zeitpunkt der Initialisierung von *ScatterUnit Testautomation* die folgenden Softwareartefakte, die dann der Reihenfolge nach auf dem jeweiligen Knoten initialisiert werden:

1. die zu testende Anwendung
2. die libScatterUnit Bibliothek
3. der für jeden Sensorknoten abhängig von seiner Rolle im Test maßgeschneiderte Testcode (z.B. Testskript, etc.)

Unmittelbar zu Beginn aktiviert ScatterUnit Testautomation jeweils die libScatterUnit Bibliothek auf jedem Sensorknoten, mit deren Hilfe der Testverlauf dezentral gesteuert wird, und beginnt dann auf die vom Sensornetz eingehenden Nachrichten zu lauschen.[47] Nun fängt die nebenläufige, auf den Sensorknoten verteilte Ausführung des Testcodes an, was nun unabhängig vom auf dem Rechner laufenden Rahmenwerk ScatterUnit Testautomation durchgeführt wird. Dies wird dadurch möglich, dass der Testcode jegliche benötigte Funktionalität lediglich durch die sich auf demselben Knoten befindenden Dienste der libScatterUnit Bibliothek findet und in Anspruch nimmt. Dadurch wird eine infrastrukturlose Steuerung und Koordination des verteilten Tests erreicht, um den Funkkanal so wenig wie möglich zu beanspruchen.

Die LibScatterUnit bietet die folgenden Dienste an:

1. Chronological synchronization service: Dieser Service ermöglicht die Synchronisation der Uhren aller Sensorknoten. Dies ist notwendig, da die Ereignisse auf den verschiedenen Sensorknoten in chronologischer Abfolge protokolliert werden müssen.
2. Setup service: Der Service ruft die Setup-Methode des Testfallcodes auf jedem Sensorknoten auf, um die notwendigen Vorbereitungen zu treffen.
3. Start service: Dieser Service startet den Testablauf auf allen Sensorknoten zur gleichen Zeit.

---

[46] Dabei macht es keinen Unterschied, ob der Testfall vollständig manuell implementiert wurde oder mit Hilfe der modellgetriebenen Visual ScatterUnit generiert wurde.

[47] Die Testergebnisse werden aus Performanzgründen nicht fortlaufend während des Testverlaufs an die Senke verschickt, sondern einmalig nach Beendigung der Ausführung des Testfalls gesendet.

4. Teardown service: Dieser Service startet die Teardown-Methode des Testfallcodes auf allen Sensoren, um nach dem Testablauf den Ausgangszustand wieder herzustellen.
5. Reporting service: Dieser Service ermöglicht unter anderem, dass die Protokolle der Sensorknoten nach dem Testablauf zur Senke gesendet werden.

Der komplette Test wird somit automatisiert durchgeführt. Der direkt auf den Sensorknoten laufende Testfallcode kontrolliert den Testverlauf mit Hilfe der *libScatterUnit*, und die auf dem Rechner laufenden Prüfroutinen sind für die Analyse des Protokolls der Testergebnisse sowie die Aufdeckung der im Test auftretenden Fehler zuständig. Insgesamt ist der effiziente Ablauf des Tests zwischen den beteiligten Sensorknoten vor allem auf die libScatterUnit Bibliothek zurückzuführen, mit der die Laufzeitumgebung der Sensorknoten erweitert wird, wodurch eine simultane Ausführung des Testobjekts und des dazu nötigen Testcodes ermöglicht wird (vgl. Abbildung 38).

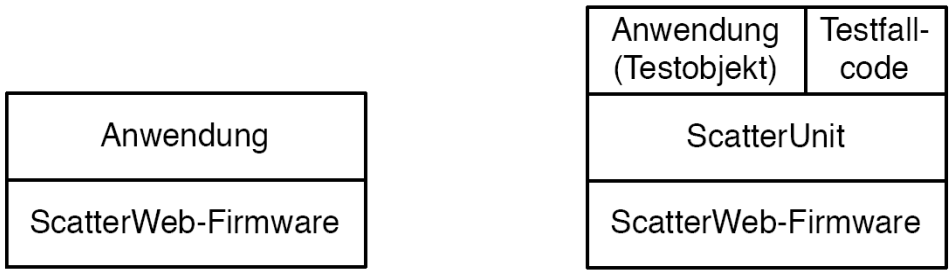


Abbildung 38 Die Laufzeitumgebung der Sensorknoten ohne (links) und mit Erweiterung (rechts) [Kam08]

Während des Testverlaufs werden alle für das Verhalten der Anwendung relevanten Ereignisse auf dem jeweiligen Sensorknoten protokolliert. Erst nach der Beendigung des Tests liest *ScatterUnit Testautomation* die Protokolleinträge von jedem Sensor und erstellt daraus ein kollektives, chronologisch geordnetes Protokoll zusammen. Nun wird dieses Protokoll mit Hilfe spezieller Prüfroutinen des Testautomation-Rahmenwerks von ScatterUnit evaluiert und entsprechende Markierungen bzw. Fehlermeldungen an den betroffenen Fehlerstellen im Protokoll eingefügt. Auf Basis dieser Markierungen werden nun die Fehler und die protokollierten Daten vom Visualisierungs-Plugin mit Hilfe der Blitz- und i-Symbole im Modelldiagramm an den entsprechenden Stellen illustriert (vgl. Abbildung 37).

Mit dem Blitz-Symbol wird es möglich, die von der Prüfroutine erzeugte textuelle Fehlermeldung dem Benutzer visuell anzuzeigen und bei Navigation der Maus auf das i-Symbol wird die aus dem Test protokollierte Information ebenfalls textuell angezeigt.

Neben den benutzerdefinierten *Log Actions* zur anwendungsspezifischen Protokollierung ermöglicht es der Protokollierungsdienst, verschiedene Ereignisse des Testverlaufs automatisch bzw. standardmäßig zu protokollieren (d.h. der Nutzer muss dafür nicht eigens eine Log Action als Modellentität definieren). Folglich werden beispielsweise der Start und das Ende (sei es eine erfolgreiche Terminierung oder ein Abbruch der Testausführung) protokolliert. Desweiteren ist es möglich, die eingehenden Datenpakete nach einem vom Benutzer im Modell konfigurierten Detaillierungsgrad zu protokollieren. So kann der Benutzer anhand des parametrisierten Modellkonstruktes selbst angeben, ob beispielsweise lediglich der Inhalt des

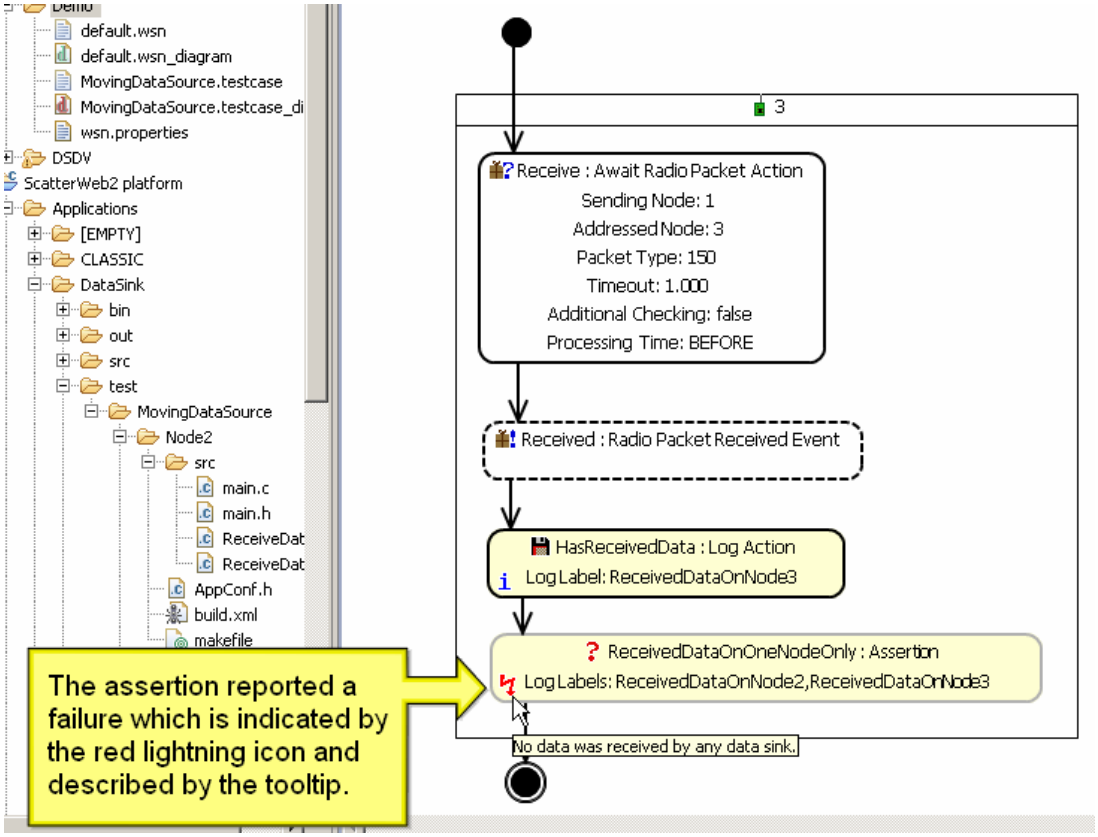


Abbildung 39 Markierungen zur Visualisierung der Testergebnisse in der Modellentität

Paketkopfes oder auch die Nutzdaten protokolliert werden sollen. Die Parametrisierung der Modellentitäten hilft damit dem Softwaretester, die Feinkörnigkeit der Protokollierung auf Modellebene auf die Anforderungen des Testfalls abzustimmen. Damit die Protokolleinträge effizient von den Prüfroutinen der ScatterUnit Testautomation bearbeitet werden können, verfügen sie über eine festgelegte Struktur. Diese sieht aus wie folgt:

1. ID des Sensorknotens, auf dem der Protokolleintrag erstellt wird
2. Der Zeitstempel, der vom Synchronisationsdienst verwaltet wird, dient zur Bestimmung des Zeitpunkts des Auftretens des mit dem Eintrag protokollierten Ereignisses, um die Protokolleinträge in eine chronologische Ordnung zu bringen.
3. Typ des Protokolleintrags zur Identifikation auf Editorebene zum Zweck der Visualisierung.
4. Anzahl aufeinander folgender Bytes in der Bytefolge
5. Bytefolge, d.h. die protokollierten Daten

Deckt die zuständige Prüfroutine anhand eines im Protokoll analysierten Log-Eintrags einen Fehler in der Anwendung auf, so wird, wie zuvor erwähnt, der betroffene Eintrag im Protokoll um eine entsprechende für den Benutzer aussagekräftige Fehlermeldung ergänzt. Programmauszug 12 zeigt einen Ausschnitt aus einem er-

weiterten Protokoll, in das eine Fehlermeldung bezüglich der Überprüfung der Nutzlast eines vom Sensorknoten mit der ID 99 empfangenen Pakets eingefügt wurde.[48] Die textuelle Fehlermeldung „The payload…“ ist durch das Blitz-Symbol im Modell einsehbar und gleiches gilt für die protokollierte Nutzlast anhand des i-Symbols. Das durch die Fehlermeldungen erweiterte Protokoll kann zusätzlich dem Benutzer textuell ausgegeben werden. Diese Variante zur Fehlernachvollziehung steht dem Benutzer zwar zur Verfügung, das Hauptanliegen des Rahmenwerks ist es jedoch, die Fehlermeldungen innerhalb der formalen Modellkonstrukte (welche die Fehler verursachen) im Editor zu visualisieren.

Gemäß den protokollierten Einträgen des Protokollausschnitts besteht die Nutzlast des Datenpakets aus den drei Bytes DE CA 00, die jedoch nicht mit der erwarteten Anzahl übereinstimmen. Aufgrund seiner Kenntnis über die zu testende Anwendung und das Testszenario kann der Benutzer in Erfahrung bringen, dass die korrekte Nutzlast des Pakets lediglich aus den zwei Bytes DE CA besteht. Daher stellen die protokollierten Daten mit ihrer zugehörigen Fehlermeldung ausreichend Informationen zum genauen Verständnis des gemeldeten Fehlers zur Verfügung.

```
[99] 9.234: 103 3 bytes data DE CA 00

FAILURE: The payload of the packet received on node 99 is corrupted.
```

Programmauszug 12 Ausschnitt eines um adäquate Fehlermeldungen ergänzten Protokolls [Kame08]

Die Daten der Protokolleinträge werden in dem EEPROM des Sensorknotens abgelegt, um bei der anschließenden Testanalyse effizient ausgelesen und bearbeitet werden zu können. Ein festgelegter Speicherbereich innerhalb des EEPROM wird ausschließlich zum Zweck der Protokollierung in Anspruch genommen, da gleichzeitig andere Segmente des EEPROM von weiteren, simultan laufenden Anwendungen benutzt werden. So wird der mit einer Speicherkapazität von 64 KB ausgestattete EEPROM neben der zu testenden Anwendung von verschiedenen Firmwaremodulen der ScatterWeb-Plattform verwendet. Freilich führt dies dazu, dass der Speicherbereich nicht grenzenlos für die Protokollierung in Anspruch genommen werden kann. Jedoch ist bei der Durchführung der Testfälle in ScatterUnit nicht von einem Langzeitbetrieb auszugehen. Normalerweise handelt es sich bei den Testfällen um Anwendungen, die für wenige Minuten ausgeführt werden und bei denen sich der benötigte Speicher kaum durch Log-Einträge aufbläht. Dank der kurzen Ausführungszeit hält sich auch der für die zu testende Anwendung benötigte Speicherbedarf zur Speicherung der Messdaten in Grenzen.[49] Ferner spielt die Größe bzw. Anzahl der Log-einträge für die Auslastung eine wesentliche Rolle. Im Fall von ScatterUnit überschreitet die maximale Größe eines Log-Eintrages in der Regel kaum 40 Bytes (wie man im letzten Programmauszug 12 sehen kann). So wurde beispielsweise das

[48] Das zugehörige Modellkonstrukt, nämlich das Prüfroutine-Log-Aktion Paar, das durch die Markierung *Nutzlast* gekoppelt wird, wurde in Abbildung 24 in Abschnitt 3.5 dieses Kapitels bereits gezeigt.

[49] Üblicherweise wird der EEPROM von der auf dem Sensorknoten laufenden Anwendung (Testobjekt) zur Beobachtung der Sensorumgebung über einen langen Zeitraum überwiegend zum Zweck der Speicherung der von der Umgebung erfassten Messdaten in Anspruch genommen. Dies ist allerdings bei der Testfalldurchführung dieser Anwendungen nicht gegeben, da die Testfälle lediglich für wenige Sekunden ausgeführt werden müssen.

DSR-Routingprotokoll [Davi01, S. 139-172] für genau eine Minute als Testobjekt ausgeführt und währenddessen sämtliche während der Ausführungszeit bei jedem Sensorknoten eingehenden Pakete durch den entsprechenden Dienst von ScatterUnit vollständig protokolliert.[50] Nach der Beendigung der Ausführung des Testfalls betrug der beanspruchte Speicheranteil des Log-Protokolls nahezu 4 Kilobytes. Eine solche Größenordnung stellt keine Einschränkung für die Testdurchführung dar. Dies belegen ebenfalls die gemessenen Daten zur Untersuchung der Speicherauslastung in 8.2 zur Evaluierung des Rahmenwerks.

4.5. Zusammenfassung

Der erste Abschnitt des Kapitels beschäftigte sich mit der Struktur der Domänensprache (DSL) zur Testautomatisierung. Hierzu stand der Aufbau des Metamodells im Zentrum. Es wurde geschildert, dass der Entwurf der spezifischen Domänensprache eine offene Struktur aufweist, welche die Testfallentwicklung nicht einengt. Einerseits steht für jeden technischen Aspekt (wie Steuerbefehle, Ereignisse und Protokolleinträge) des automatisierten Tests jeweils eine korrespondierende Modellentität in der DSL zur Verfügung, deren zugehöriges Codeartefakt vom Nutzer gemäß der gerade zu testenden Anwendung manuell angepasst werden kann. Andererseits bleibt die Option offen, durch strukturierte Erweiterung des Metamodells häufig verwendete Konstrukte in der DSL fest zu verankern, so dass deren Code voll automatisiert generiert werden kann. Ein Diagramm, welches das gesamte Metamodell veranschaulicht, ist im Anhang 11.4 zu finden.

Im zweiten Abschnitt wurde geschildert, wie die Validierung der mit Hilfe der DSL erstellten formalen Modelle vonstatten geht. Um die Qualität des generierten Codes und damit des Testfalls sicherzustellen, wird das Testfallmodell hinsichtlich seiner syntaktischen und semantischen Regeln in Form von Bedingungen überprüft. Dies erfolgt nicht nur, wenn das Testfallmodell generiert wird, sondern auch während der Modellierung des Testfalls. Da die Modellvalidierung mit Hilfe des openArchitectureWare-Rahmenwerks erfolgt und der visuelle Editor mit Hilfe des Graphical Modelling-Rahmenwerks erstellt wurde, werden die Komponenten mit Hilfe des GMF2 Adapters kombiniert, um die Live-Validierung zu ermöglichen. Zu diesem Zweck enthält der visuelle Editor den GMF2 Adapter, der den Zugriff auf die model checking engine von openArchitectureWare ermöglicht. Diese wird wiederholt aufgerufen, um das Testfall-Modell, das gerade editiert wird, auf der Basis von Bedingungen zu validieren. Eine große Anzahl von syntaktischen Regeln kann kompiliert werden, der wahre Nutzen der Modellvalidierung in Visual ScatterUnit entfaltet sich jedoch bei der Prüfung semantischer Regeln. Im Kapitel wurden repräsentative semantische Randbedingungen erläutert und diskutiert, wobei gezeigt wurde, wie stark sich dadurch die Robustheit des verteilten Tests steigert. Insgesamt sichert die Modell-Validierung die Qualität des auszuführenden Testfalls und erhöht damit die Robustheit der Testfallentwicklung. Weiterhin zeigt die Live-Validierung Fehler während der Testfallmodellierung an, so dass der Benutzer diese zeitnah korrigieren kann.

---

[50] Es handelt sich um den von ScatterUnit standardmäßig unterstützten Protokollierungsdienst für die empfangenen Pakete, bei dem der Detaillierungsgrad der Protokollierung (Paketkopf, usw.) vom Benutzer auf Modellebene beliebig konfiguriert werden kann. Siehe hierzu Abbildung 33 in 4.1.

Als nächstes wurden der Codegenerierungsprozess und dessen Realisierung erläutert. Hier lag u.a. ein Augenmerk auf der Zusammenfügung von automatisiert generiertem und manuell erstelltem Code. Dabei wurde der Gebrauch der Technik der geschützten Bereiche bei der Codegenerierung erläutert. In den bisherigen Abschnitten werden die Phasen des modellgetriebenen Prozesses deutlich, angefangen von der Metamodellierung bis hin zur Generierung der Codeartefakte der Dienste zur koordinierten Durchführung des verteilten Testes auf den verschiedenen Sensorknoten.

Die vielfältigen Funktionalitäten dieser Dienste wurden im darauffolgenden und letzten Abschnitt beschrieben. ScatterUnit bietet viele Dienste, die für den Test verschiedener Anwendungen wiederholt in Anspruch genommen werden. So kann z. B. die Messung von Sensordaten oder die Topologie eines Sensornetzwerks simuliert werden. Auf diese Weise können sich oft wiederholende Testdurchläufe von ScatterWeb-Anwendungen unabhängig von der aktuellen Umgebung des Sensor Netzwerks ausgeführt werden. Ferner wurde geschildert, auf welche Weise es die Architektur ScatterUnits ermöglicht, dass das Verhalten der getesteten Anwendung von der Erweiterung durch ScatterUnit unbeeinflusst bleibt.

5. ScatterFactory: Der Pfad zur Anwendungsentwicklung

ScatterFactory [Hent06] stellt den zweiten Teil der generativen Infrastruktur der ScatterClipse-Werkzeugkette dar. Analog zu Visual ScatterUnit ist ScatterFactory ein modellgetriebenes Rahmenwerk, das mit denselben Werkzeugen und Technologien wie Visual Scatter-Unit realisiert wurde. Dies ermöglicht auch ihre problemlose Zusammenarbeit, die später behandelt wird. Mit Hilfe von ScatterFactory werden Software-Familien für die auf den Embedded Sensor Boards (ESBs) laufenden Anwendungen automatisiert erstellt. Abbildung 40 zeigt die Architektur des ScatterFactory-Rahmenwerks, die veranschaulicht, welche Werkzeuge und Technologien zur Realisierung eingesetzt wurden und wie diese zusammen arbeiten. Ferner werden auf der rechten Seite der Abbildung die in jeder Stufe des Entwicklungsprozesses entstehenden bzw. benötigten Dateien aufgeführt. Im linken Balken werden zu jeder Prozessphase die involvierten Werkzeuge bzw. Plugins angegeben.

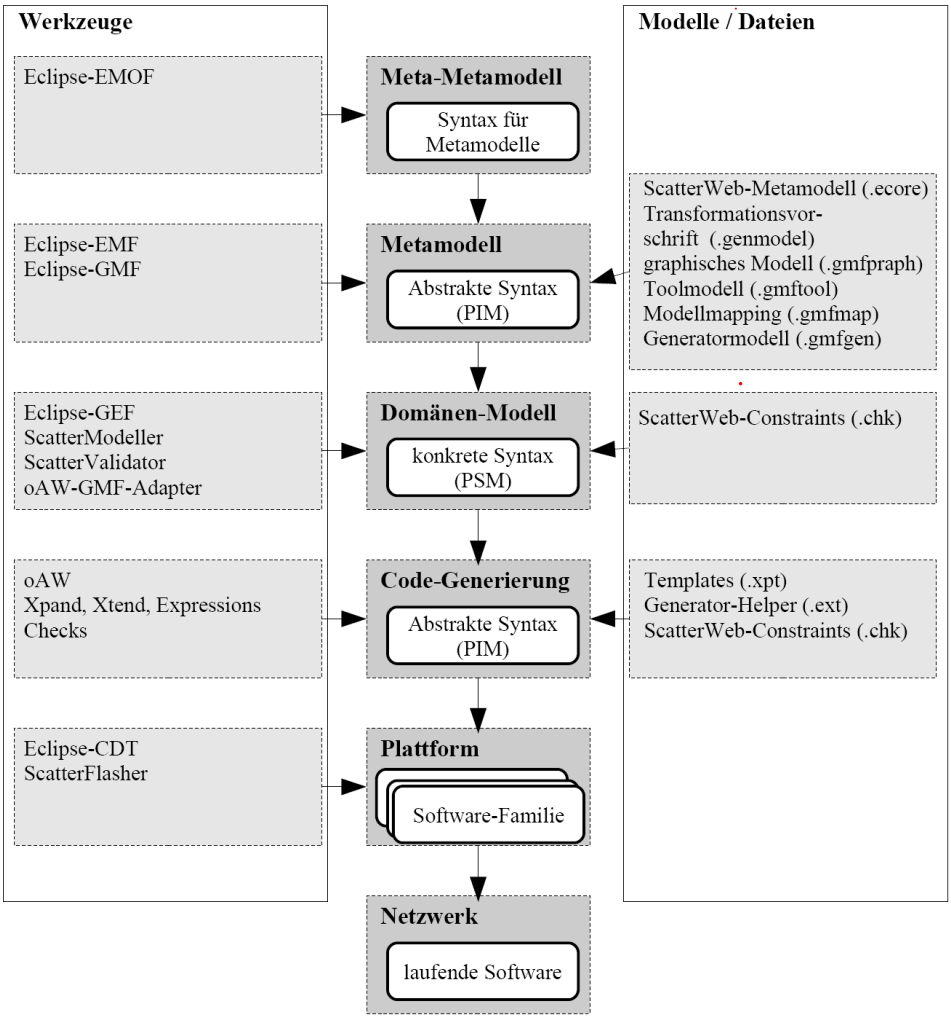


Abbildung 40 Die Architektur von ScatterFactory [Hent06]

5.1. Das Metamodell

Die Entwicklung des Metamodells unterliegt – anlog zu Visual ScatterUnit –einem iterativen Prozess. Demzufolge wurde das Metamodell entsprechend dem Charakter der ScatterWeb-Plattform graduell aufgebaut. Abbildung 41 veranschaulicht die Elemente des Metamodells sowie die zwischen ihnen bestehenden Beziehungen. Hierbei wurde UML als Notation verwendet.

Ausgangspunkt bei der Modellierung ist die Netzwerkinstanz (*Network*), die das gerade zu modellierende WSN mitsamt seinen Knoten repräsentiert. Daher bildet das *Network* Element die Wurzel des Metamodells. Gemäß dem Metamodell besteht ein drahtloses Netzwerk aus[51] den Sensorknoten *Sensor*, den auf den Knoten laufenden Softwarecontainern *Container* und den entsprechenden Relationen *Connections*, welche die verschiedenen Modellinstanzen logisch verknüpfen. Folglich besteht eine Aggregation zwischen diesen drei Metamodellelementen einerseits und der *Network*-Klasse andererseits, da die Darstellung eines durch das Modellelement repräsentierten WSNs lediglich aus der Modellierung der bereits genannten drei Modellinstanzen besteht.

Das Hauptanliegen beim Entwurf des Metamodells ist die Konstruktion einer DSL, deren Ausdrucksmächtigkeit die Repräsentation der auf den einzelnen Sensorknoten laufenden Softwaremodule sowie die zwischen ihnen bestehenden Abhängigkeiten erfasst. Solche Abhängigkeiten orientieren sich stark an den verwendeten Firmwaremodulen des Sensorknotens. Daher fußt die Schlüsselidee bei der Darstellung der einzelnen Module auf dem Komponentenprinzip.[52] Ein Container setzt sich aus einer Menge von Komponenten zusammen, wobei jede Komponente mehrere voneinander abhängige Firmwaremodule umfasst. Diese gegenseitigen Abhängigkeiten werden gemäß der Softwarekomponententechnik anhand von *Ports* ausgedrückt. Dabei wird zwischen *Required Ports* und *Provided Ports* unterschieden. Bei ersteren wird angegeben, welche Funktionalitäten für eine Komponente von einer anderen benötigt werden. Sämtliche von einer Komponente für andere angebotenen Dienste werden mittels *Provided Ports* spezifiziert. Da ein Port entweder *provided* oder *required* sein kann, wurde dies im Metamodell durch eine Aggregation angegeben. Ferner können gemäß dem Metamodell für eine Komponente verschiedene Versionen existieren. Daher ist es möglich, dass für denselben Komponententyp unterschiedliche Implementierungen existieren, die je nach der laufenden Anwendung bzw. dem Sensortyp vom Benutzer untereinander ersetzt werden können. Ein Beispiel dazu findet sich in 5.4.

---

[51] Das verbale „besteht aus" wurde anhand der korrespondierenden Aggregation im Metamodell realisiert.

[52] „In der angewandten Informatik ist die komponentenbasierte Entwicklung (engl.: Component Based Development – CBD) ein aus früheren Ansätzen entwickeltes und um deren Schwachstellen bereinigtes Paradigma. Grundlage dieses Konzeptes sind Software-Komponenten, die die Wiederverwendbarkeit von Software-Artefakten verbessern sollen. […] Der Grundgedanke komponentenbasierter Entwicklung ist die Unterteilung von Anwendungen in wiederverwendbare Komponenten, um möglichst wenig Code neu programmieren zu müssen. Mit der Zeit kann so ein ‚Komponentenmarktplatz' entstehen, aus dem heraus Anwendungen nach dem Baukastenprinzip zusammengestellt werden. Zusätzliche Komponenten müssen nur für Funktionalität entwickelt werden, für die es bisher keine Implementierung gibt." [WebComp]

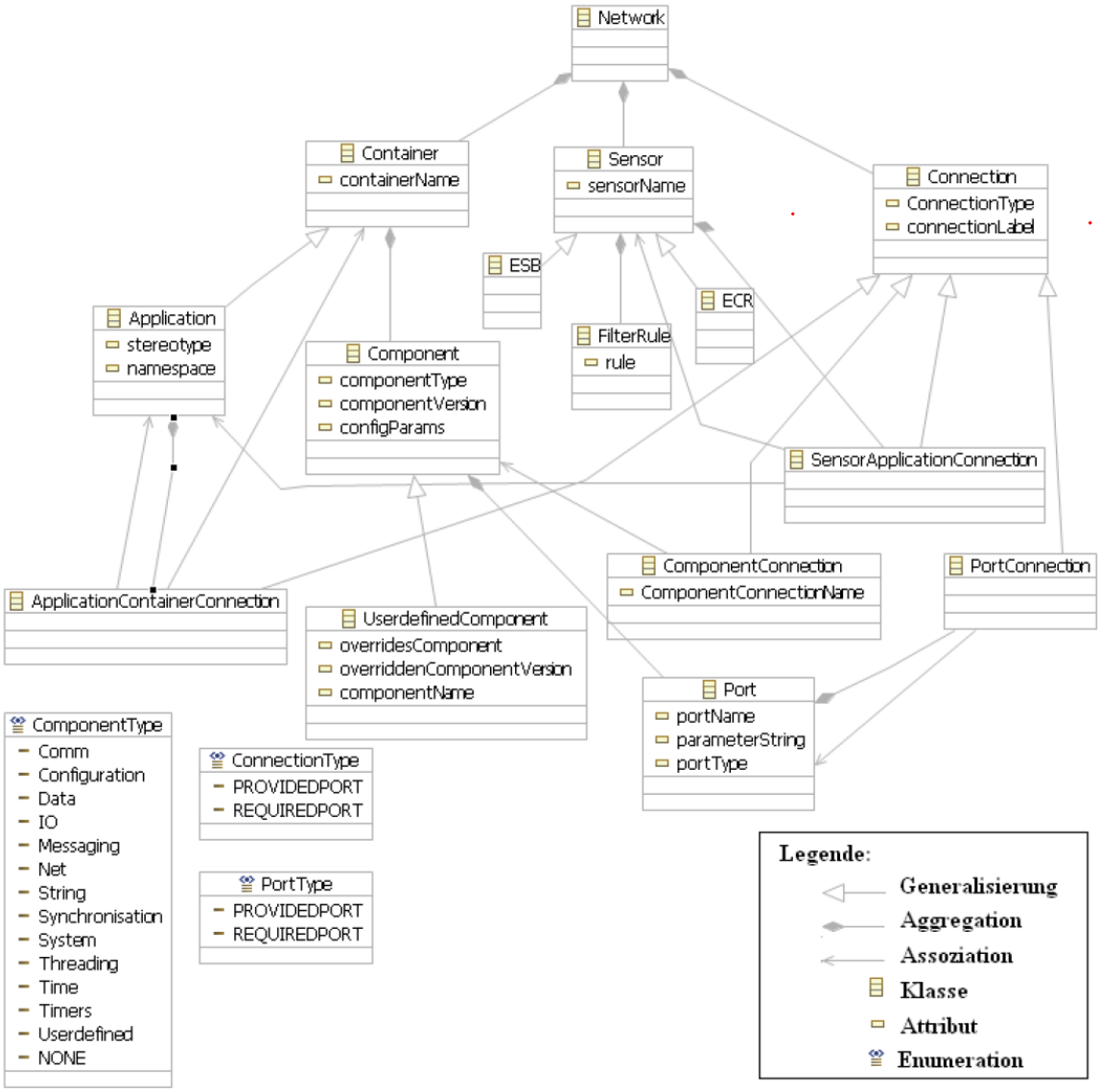


Abbildung 41 Das Metamodell von ScatterFactory [Hent06]

Betrachtet man die gesamte Struktur der Software eines einzelnen Sensorknotens, so unterscheidet man dabei zwischen zwei Ebenen: der auf dem Knoten laufenden Anwendung und den dafür benötigten Firmwaremodulen. Letzere werden, wie zuvor geschildert, durch die Definition bzw. die Verwendung passender Container im Modell zusammen gefügt. Für die Modellierung der Anwendung steht die Klasse *Application* zur Verfügung, die als Verknüpfung zwischen dem Sensor und den darauf installierten Firmwarecontainern fungiert. Die Anwendung stellt den Bereich in der modellierten Software dar, in dem der Entwickler den Code manuell implementiert. Dafür werden zur automatisierten Codegenerierung auf Template-Ebene die in 4.3 erläuterten geschützten Bereiche (*Protected Regions*) eingesetzt. Allerdings kann der Code einer Anwendung zunächst einmalig entwickelt und daraufhin in korrespondierenden Templates abgelegt werden, was eine Wiederverwendung des Codes ermöglicht. Um auf die zugehörigen Templates einer Anwendung während der Codegenerierung zugreifen zu können, verfügen die Anwendungen über eindeutige Bezeichner und zwar den Namen (angegeben durch das Attribut *namespace* im Metamodell) und eine mögliche zusätzliche Spezifikation wie beispielsweise die Version. Dies wird im Metamodell durch das Attribut der *Application*-Klasse *Stereotype* angegeben.

5.2. Die Modellierungsebene

Das Herzstück der Modellierung in ScatterFactory ist der generative Editor, in dem die Metamodellelemente auf eine übersichtliche Weise wiedergegeben werden. Dies ermöglicht dem Benutzer eine adäquate Zusammenstellung der Modellkonstrukte zur grafischen Anwendungserstellung. Abbildung 42 zeigt einen Bildschirmabzug des generativen Editors und seine wesentlichen Elemente.

In der Abbildung sieht man unter 1 die Palette, in der sämtliche Elemente des Metamodells zur Auswahl durch den Benutzer aufgelistet sind. Des Weiteren werden die aufgelisteten Elemente gemäß ihrer Kategorisierung im Metamodell in der Palette aufgeteilt. So befinden sich in der oberen Kategorie alle Modellelemente zur Wahl einer Sensorinstanz.[53] In der mittleren Gruppe stehen die Softwaremodellinstanzen zur Auswahl und zwar von der Anwendung hin bis zu den Ports. In der unteren Gruppe finden sich alle Arten von Verknüpfungen, durch die die Modellelemente logisch verbunden werden. Ferner kann durch die Zoomfunktion die Größe des Modells in der Modellierungsfläche vom Benutzer beliebig eingestellt werden. Alternativ zur Palette können Modellelemente über dem Kontextmenü (siehe 2) direkt ins bereits bestehende Modell in der Modellierungsfläche eingefügt werden. Das Menü wird angezeigt, sobald der Benutzer zur Modellierungszeit mit der Maus über das Modelldiagramm navigiert. Diese Funktionalität dient der besseren Handhabung bei der Zusammenstellung der Modellkonstrukte während der Modellierung. In 4 ist das *Problem View* des Eclipse-Rahmenwerks zu sehen, in dem die gemäß dem Metamodell definierten Attribute eines Modellelements angezeigt werden, wenn das betroffene Modellelement im Diagramm in der Editorfläche ausgewählt wird.

---

[53] Die MSBs kommen hier noch nicht in Betracht, da das gesamte Rahmenwerk hinsichtlich der neuen Generation der MBS-Sensorknoten optimiert wurde, was später in diesem Kapitel in einem eigenen Abschnitt ausführlich behandelt wird.

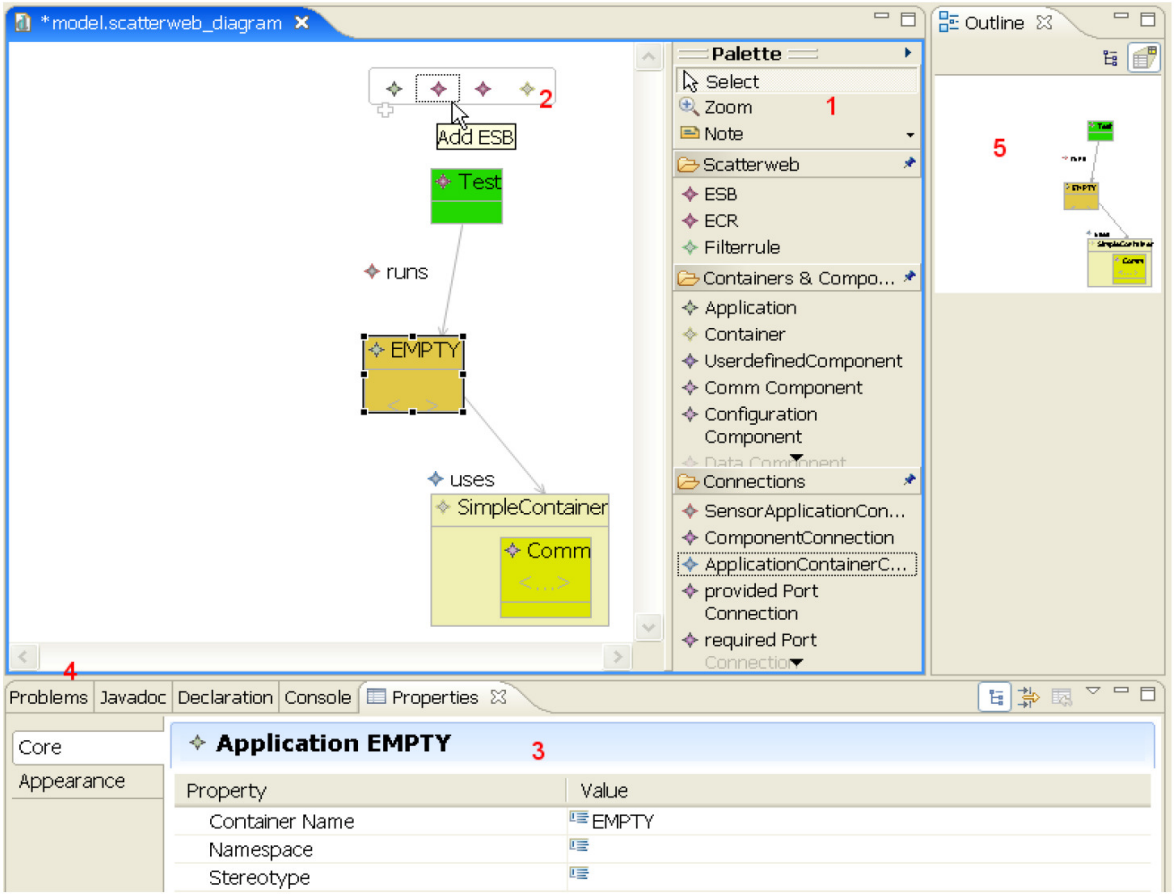


Abbildung 42 Der ScatterFactory Editor [Hent06]

Somit kann der Benutzer neben der grafischen Modellierung der Elemente zugleich deren Attribute konfigurieren. Die Zusammenarbeit zwischen dem Editor und der Eclipse-Problem View zeigt einen weiteren Vorteil der Eclipse-Integration des Rahmenwerks, das in der ScatterClipse-Werkzeugkette verwendet wird. Werden die definierten Modelle groß und somit unübersichtlich, so wird in 5 stets eine Gesamtübersicht des Modelldiagramms angezeigt, in dem der Benutzer einen bestimmten Teil selektieren kann, wodurch er zu diesem Teil in der Modellierungsfläche des Editors direkt gelangt.

Der Editor stellt etliche Funktionalitäten zur Verfügung, die den Benutzer in die Lage versetzen, durch überschaubare Interaktion aussagekräftige grafische Modelle auf Basis des zuvor definierten Metamodells zu bilden. In der Abbildung 43 wird ein mit Hilfe des Editors erstelltes Modellierungsbeispiel einer konkreten Anwendung gezeigt. YellowSwitcher ist eine typische Anwendung auf einem Sensorknoten, da sie ein gängiges Szenario der WSN-Domäne realisiert. Der Sensorknoten versendet und empfängt Pakete im WSN. Des Weiteren nimmt der Knoten Überwachungsaktivitäten im WSN wahr, indem er auf bestimmte Ereignisse im Netz durch die Aktivierung entsprechender Funktionalitäten reagiert. So wird durch die YellowSwitcher-Anwendung die gelbe Lampe ein- und ausgeschaltet (toggled), sofern ein bestimmtes Paket vom Knoten empfangen wird. Dieses Szenario fungiert als repräsentatives Beispiel für den Anteil an benötigtem Code, der in ScatterFactory automati-

siert generiert wird[54] und durch den die Realisierung einer herkömmlichen Aufgabe eines Sensorknotens im WSN ermöglicht wird. Wie erwähnt, fungiert die Anwendung als Bindeglied zwischen der Sensorinstanz und dem Container. In diesem Beispiel beinhaltet der Container sechs Firmwaremodule (von insgesamt 11), welche für die Anwendung benötigt werden. Bei der Instanz des Sensors sind zwei Spezifikationen angegeben „USE_DUMMIES“ und „NO_COMMANDS“. Diese sind sogenannte Filterregeln zur Optimierung des Codegenerierungsprozesses hinsichtlich der gerade modellierten Anwendung. Sie sorgen dafür, dass während der Codegenerierung unnötige Codesegmente für die jeweilige modellierte Anwendung innerhalb der Templates herausgefiltert werden, insbesondere solche der im Quellcode der Firmwaremodule verstreut eingeflochtenen Steuerbefehle. Insofern fungieren die Filterregeln als Technik, welche die Herausforderung der Codevermischung im Hinblick auf eine optimierte Codegröße der modellierten Anwendung auf Template-Ebene elegant löst. Dieser Aspekt wird in Abschnitt 6.4 bei der Betrachtung des Codegenerierungsprozess ausführlich behandelt.

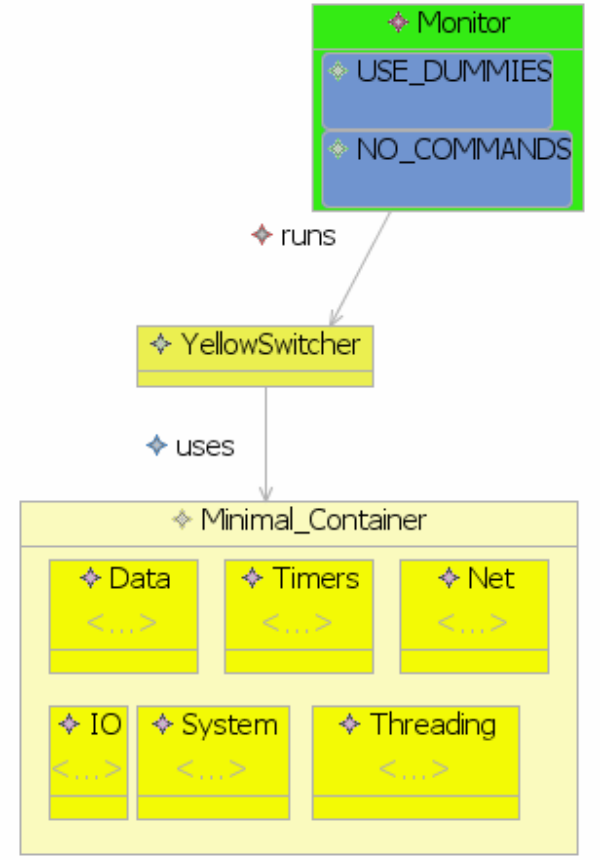


Abbildung 43 Modellierungsbeispiel einer Anwendung [Hent06]

[54] Die genaue Größe zum benötigten Code im Vergleich zu anderen Anwendungen wird in 8.1 bei der Evaluierung angegeben.

5.3. Die Modellvalidierung

Analog zu Visual ScatterUnit basiert die Livevalidierung in ScatterFactory auf der Zusammenarbeit zwischen dem oAW-Generator und dem EMF/GMF-Editor, die durch den oAW-GMF Adapter (vgl. 2.5) erreicht wird. Ebenfalls wie bei Visual ScatterUnit erfolgt die Formulierung der Randbedingungen mittels der vom oAW-Rahmenwerk unterstützten Checks-Sprache. Für ScatterFactory wurde das Plugin ScatterValidator entwickelt, um die definierten Checks-Ausdrücke mit dem Editor zu verbinden. Hierzu erweitert das Plugin mittels der Klasse ScatterWebValidateAction den Erweiterungspunkt (Extension Point) des EMF-Metamodells (org.eclipse.emf.-validation.constraintProviders) um den oAWConstraintsProvider. Damit stellt die Klasse die notwendigen Methoden zur Verfügung, um dem EMF/GMF-Editor einen Zugang auf die im Ordner scatterweb.chk abgelegten Checks-Ausdrücke zu verschaffen, damit diese gegen jegliche Modelländerungen im Editor überprüft werden können.

Konzeptionell handelt es sich bei den spezifizierten Checks-Ausdrücken um zwei Kategorien. Bei der ersten befassen sich die Ausdrücke mit der Überprüfung von syntaktischen Bedingungen der Domänensprache. Mit anderen Worten, die Zielsetzung ist es, zu gewährleisten, dass die Modellelemente richtig editiert werden. Bei der zweiten Klasse – in der Regel die umfangreichere – der Checks-Ausdrücke geht es um semantische Aspekte der Domäne. Hierbei wird die Modellierung gegen domänenspezifische Randbedingungen überprüft. Programmauszug 13 zeigt einen Ausschnitt einiger Checks-Ausdrücke der ScatterFactory:

```
//Editing constraints
context Sensor ERROR " Sensors schould have a name " :
        sensorName.compareTo(null) >0 ;
context Sensor ERROR " Sensors need an application " :
        sensorApplicationConnection.compareTo(null)>0;
context Application ERROR " Applications need to be connected to at least one Container " :
        applicationContainerConnections.size.compareTo(0)> 0;
context Application ERROR " Applications need a Name " :
        containerName.compareTo(null)>0;
context Container ERROR " Containers must contain at least one Component " :
        components.size.compareTo(0)>0;
context UserdefinedComponent ERROR " User defined components need to be named " :
        componentName.compareTo(null) >0;
context Port ERROR "Ports cannot be provided and required at the same time" :
        !(portConnections.select(e|e.ConnectionType.toString().matches("PROVIDEDPORT")).to
        .intersect
        (portConnections.select(e|e.ConnectionType.toString().matches("REQUIREDPORT")).toPort).size>=2));

//domain specific constraints
context Port ERROR " Port \"Actor\" need not be connected to port defined in the Timers-module" :
        portName.matches("Actor")&& portConnections.exists(e|e.toPort.portName.matches("AddTimers")
                                                        ||e.toPort.portName.matches("ModifyTimers"));
context Port ERROR " Port \"Events\" must be connected to ports defining addind and modification of timers"
        portName.matches("Events")&& !(portConnections.exists(e|e.toPort.portName.matches("AddTimers")
                                                        ||e.toPort.portName.matches("ModifyTimers")));
```

Programmauszug 13 Einige Checks-Ausdrücke in ScatterFactory: syntaktische Randbedingungen (oben) und semantische domänenspezifischen Randbedingungen (unten) [Hent06]

Im Folgenden wird aus jeder Klasse jeweils ein Ausdruck näher betrachtet, um ein genaueres Bild des Einsatzes der Checks-Sprache für ScatterFactory zu gewinnen. Betrachtet man den zuerst spezifizierten Ausdruck, so sieht man, dass der Ausdruck der Überprüfung dient, ob jede Sensorinstanz über einen eindeutigen Namen verfügt und der Name kein leeres String ist. Hier ist leicht erkennbar, dass der Ausdruck sich mit einer syntaktischen Fragestellung der Domänensprache befasst. Beim ersten, recht komplexen Ausdruck der unteren Kategorie handelt es sich im Gegensatz dazu um einen domänenspezifischen Ausdruck. Dieser zeigt einen Modellierungsfehler an im Falle, dass der für eine passive Komponente definierte Port Actor mit einem weiteren Port verknüpft wird, welcher einen Dienst des ereignisgesteuerten Timermoduls zur Verfügung stellt. Durch diesen Ausschluss wird bewirkt, dass kein unnötiger Code für eine solche passive Komponente generiert wird, was zu einer Optimierung des Speicherbedarfs führt. Eine bestmögliche Verknüpfung der Komponenten richtet sich dank entsprechender Modellüberprüfung nach den Eigenschaften der Sensorknoten und den darauf laufenden Softwaremodulen.

5.4. Die Codegenerierung

Durch die Integration des oAW-Codegenerators wie auch des GMF-Editors als Plugins in das Eclipse-Rahmenwerk kann nach der Erstellung und Validierung des formalen Modells der Codegenerierungsprozess innerhalb der Eclipse-Umgebung angestoßen werden. Hierzu muss (wie Abbildung 44 veranschaulicht) durch den Befehl *start_generator.oaw* und dann *Run As* im Projektexplorer von Eclipse der Generator mittels *oAW Workflow* aktiviert werden. Gemäß der Konfiguration der Workflow-Datei des oAW-Generatorrahmenwerks wird das formale Modell aus dem Editor vom Codegenerator erkannt und anschließend serialisiert, um dann zwecks Codegenerierung die den serialisierten Modellinstanzen korrespondierenden Templates einzubinden.

Sämtliche aus dem Generierungsprozess entstandenen Benachrichtigungen werden in der *Console View* von Eclipse dem Benutzer mitgeteilt, wodurch er das Voranschreiten bzw. den Ausgang des Generierungsprozesses nachvollziehen kann. Die generierten Artefakte werden in einem speziellen Zielordner namens *src-gen* abgelegt (vgl. Abbildung 45). Dieser Zielordner kann beliebig vom Benutzer innerhalb der Workflow-Datei des Codegenerators bestimmt werden. Bei den generierten Artefakten handelt es sich um Quellcodedateien (wie die C-Dateien und deren Kopfzeilen für die Firmwaremodule und die Anwendung) sowie Konfigurationsdateien (wie Make-Dateien und XML-Dateien für das Ant-Build), um die anschließende Kompilierung und Ausbringung (Deployment) der Software auf die Sensorknoten zu bewerkstelligen.[55]

Im Codegenerierungsprozess werden zwei für die Optimierung der Codegröße entscheidende Techniken verfolgt. Die erste ist die Verwendung von Dummy-Templates. Dabei werden bei der Generierung der im Modell nicht vorkommenden und somit nicht benötigten Firmwaremodule Dummy-Templates eingebunden. Ein Dummy-Template beinhaltet lediglich die Signaturen der Prozeduren dieses Firmwaremoduls, wobei der Rumpf jeder Prozedur leer bleibt (daher die Bezeichnung Dummy). Somit wird gewährleistet, dass am Ende des Generierungsprozesses ein

[55] Mehr zu den Aktivitäten der Deployment-Phase im folgenden Abschnitt über das Entwicklungs-Plugin.

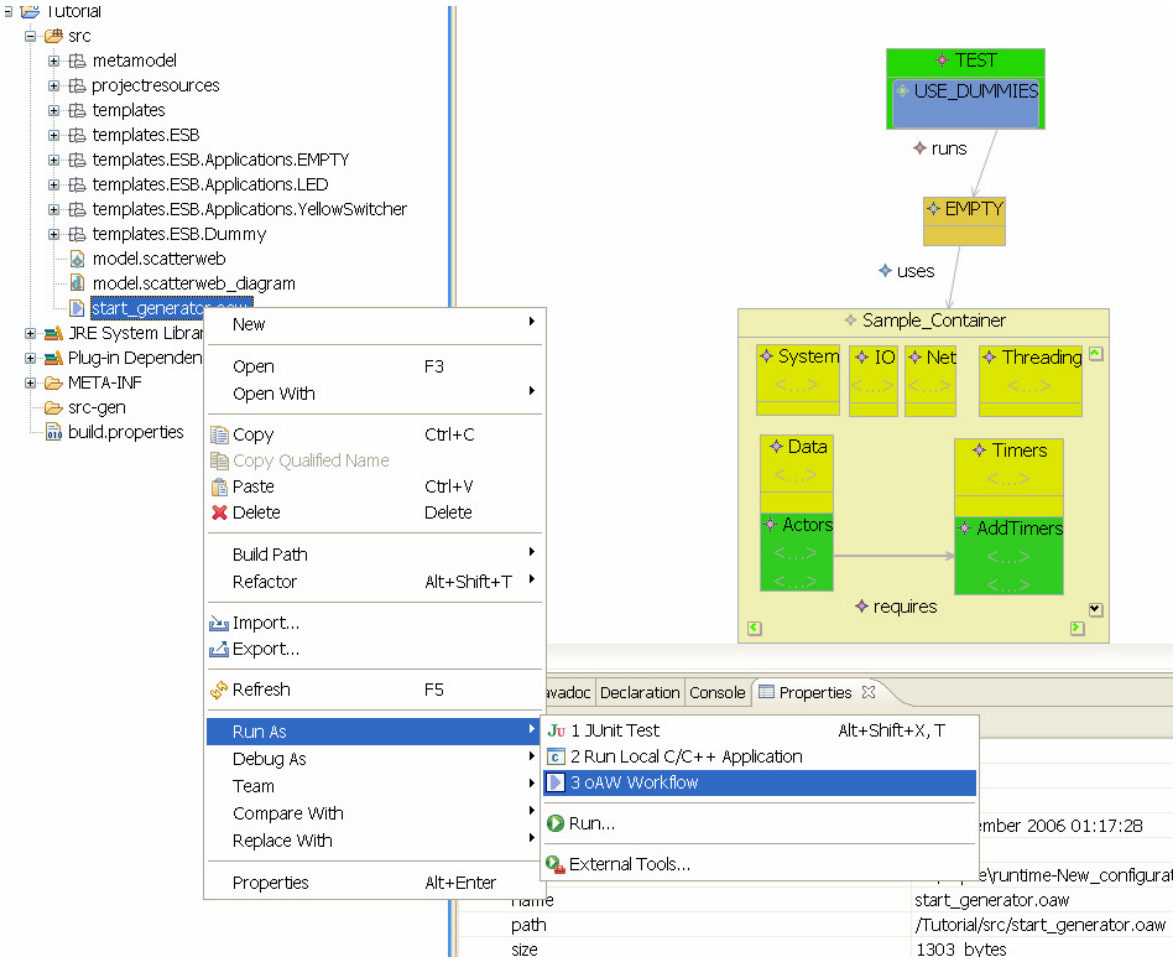


Abbildung 44 Gleichzeitiger Ablauf des Modellierungs- und des Generierungsprozesses innerhalb von Eclipse [Hent06]

korrekt kompilierter Code steht, wobei unnötiger Code größtenteils vermieden wird.[56] Um die Funktionalität von Dummy-Templates bei der Modellierung zu aktivieren, muss die Anweisung USE_DUMMIES (vgl. Abbildung 43) bei der Sensorinstanz angegeben werden.

Die zweite Technik besteht im Einsatz von Filterregeln, mit deren Hilfe während der Codegenerierung gemäß der jeweiligen im Modell spezifizierten Filterregeln bestimmte Codeartefakte bei der Emittierung des betroffenen Templates herausgefiltert werden. Dies sind insbesondere überflüssige Codesegmente der im Quellcode der Firmwaremodule verstreut eingeflochtenen Macros und Konsolekommandos. Es wird zwischen drei Arten von Filterregeln unterschieden, wobei jede die Filterung spezifischer Codeartefakte bewirkt.

[56] In 8.1 wird die erzielte Effizienz des Einsatzes von Dummy Templates anhand der ersparten Codegröße bezüglich repräsentativer Anwendungsszenarien ausführlich diskutiert.

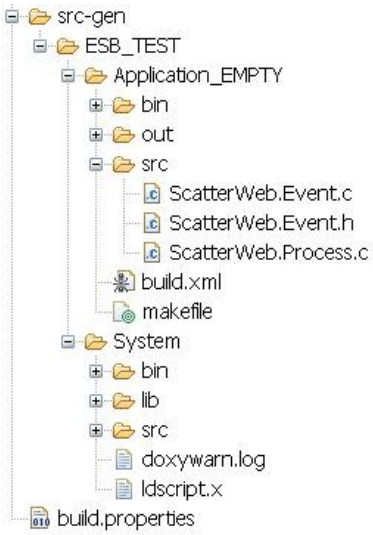


Abbildung 45 Die generierten Artefakte in ScatterFactory [Hent06]

- NO_COMMANDS: Hierdurch werden sämtliche auf Konsolekommandos bezogene Codezeilen im Template bei der Einbindung des Codes nicht in Betracht gezogen.
- NO_SERIAL: Diese Filterregel bewirkt, dass die Kommandozeilen für die serielle Schnittstelle bei der Einbindung des Codes nicht herausgefiltert werden (die übrigen Kommandozeilen werden herausgefiltert).
- MIN_COMMANDS: Nur signifikante, oft verwendete Kommandos wie das Setzen der ID werden eingebunden. Dadurch wird ein großer Anteil von Kommandozeilen herausgefiltert.

Die Realisierung der Filterregeln in den Xpand Templates des oAW fußt auf der Verwendung von Erweiterungen (Extentions), die in den Templates unterstützt werden. Ihre Technik wurde in 2.5 erörtert. Solche Erweiterungen werden überall innerhalb des Templates aufgerufen und zwar unmittelbar bevor eine betroffene zu filternde Kommandozeile aufgefunden wird. Die aufgerufene Erweiterung bekommt hierfür den Namen der Filterregel als formalen Parameter. Im Programmauszug 14 wird ein Ausschnitt eines Templates gezeigt, in dem die Filterregel NO_SERIAL verwendet wird:

```
<< IF !containsCommandFilter(Sensor) >>
<< IF !sensor.filterRules.rule.exists(e | e.toString().matches("NO_SERIAL")) >>
Packet_t packet
Packet.to = String_parseInt(&str[], Null); packet.type = PING_PACKET;
…
…
```

Programmauszug 14 Einsatz von Filterregeln bei der Codegenerierung [Hent06]

Die Korrespondenz zwischen der Modellierung einerseits und dem damit verbundenen Generierungsprozess auf der anderen Seite erfolgt bei der Anwendungsentwicklung in ScatterFactory reibungslos. Dies zeigt das folgende Szenario, das illustriert, wie jede beliebige Änderung des Modells von dem darauf folgenden Codegenerierungsprozess mittels entsprechender Anpassung der Templates ohne weiteres unterstützt werden kann.

Im Programmauszug 15 wird anhand eines Xpand Templates veranschaulicht, wie eine zur Standardkomponente des Firmwaremoduls Net alternative Implementierung derselben Komponente bei der Codegenerierung verwendet wird. Bei der alternativen Implementierung handelt es sich um die auf CSMA/CA[57] basierende Realisierung des Net-Firmwaremoduls. Auf Modellebene wird das Attribut ComponentVersion der Komponenteninstanz (siehe dazu Abbildung 41 des Metamodells) auf CSMA/CA vom Benutzer initialisiert, was über das Properties View von Eclipse bewerkstelligt werden kann (vgl. Abbildung 42 Teil 3). Dadurch wird bekanntgegeben, dass es sich bei der Komponente um die alternative CSMA/CA-Implementierung des Net-Firmwaremoduls handelt. Der obere Ausschnitt des für das Net-Firmwaremodul stehenden Templates (Component.xpt) zeigt, wie auf Codegenerierungsebene beim Durchlaufen des Templates abgefragt wird, ob es sich bei der Belegung des ComponentVersion-Attributs um den Wert CSMA/CA handelt oder nicht. Ist dies der Fall, so wird der zugehörige Code eingebunden.

```
<<IF comp.componentType.toString().matches("Net")>>
    <<IF comp.componentVersion==null>>
        <<Expand ESB::ScatterWebNetc::file(sensor, comp) FOR comp>>
        <<Expand ESB::ScatterWebNeth::file(sensor, comp) FOR comp>>
    <<ENDIF>>
    <<IF comp.componentVersion=="CSMACA">>
        <<Expand ESB::ScatterWebNetCSMAc::file(sensor, comp) FOR comp>>
        <<Expand ESB::ScatterWebNetCSMAh::file(sensor, comp) FOR comp>>
    <<ENDIF>>
<<ENDIF>>
```

Programmauszug 15 Generierung eines alternativen Firmwaremodules [Hent06]

[57] CSMA/CA ist die Abkürzung für Carrier Sense Multiple Access - Collision Avoidance. "CSMA/CA vermeidet Kollisionen durch Verwendung eines Prioritätenschemas, bei dem das Zugriffsrecht von den Stationen signalisiert wird. Ist eine Station sendebereit, horcht sie das Übertragungsmedium für eine bestimmte Zeit ab. Im Falle von Funk-LANs ist das Übertragungsmedium der Funkkanal, der für eine bestimmte Zeit, das Inter Frame Spacing (IFS), abgehört wird. Nach Ablauf der IFS sendet die Station bei freiem Medium eine Sendeanforderung (RTS). Dieses Signal wird von allen anderen Stationen empfangen, die daraufhin eventuelle Sendewünsche zurückstellen. Die Empfangsstation sendet eine Sendebereitschaft (CTS), woraufhin die Stationen die Daten übertragen können. Das Ende der Übertragung wird durch ein ACK-Signal signalisiert. Das kollisionsfreie Verfahren eignet sich besonders für Übertragungen auf Medien, die nicht auf Kollisionen reagieren, wie z.B. Funkkanäle." [WebCSMA/CA]

5.5. Kompilierung und Initialisierung der Software (Deployment)

Abschließend werden die aus den Templates generierten Codeartefakte kompiliert, um dann auf die Sensorknoten ausgebracht zu werden. Zu diesem Zweck wurde das ScatterFlash-Plugin entwickelt (vgl. Anhang 11.3.2), das die notwendigen Funktionalitäten zur Verfügung stellt. Die Realisierung des Plugins basiert auf CDT (Eclipse C/C++ Development Tools) [WebCDT], da der generierte zu kompilierende Quellcode reiner C-Code ist. Somit kann der Quellcode der Sensorknoten mit Hilfe der CDT innerhalb der Eclipse-Umgebung ohne weiteres editiert werden. Die Steuerung des Kompilierungsprozesses wird mit Hilfe des in Eclipse standardmäßig integrierten ANT-Plugins erledigt. Ferner können die kompilierten Softwareartefakte (Hex-Files) mit Hilfe des ANT-Plugins auf die Sensorknoten ausgebracht werden.

Die Kompilierung der Softwareartefakte sowie ihre serielle Ausbringung (serial Flashing) auf die Sensorknoten erfolgt mit Hilfe der mspgcc-Software [Webmspgcc]. Die Steuerung dieser Software zur Aktivierung der gewünschten Funktionalitäten beruht auf der Verwendung der Makefiles-Technik. Die zu diesem Zweck bereitgestellten, ebenfalls automatisiert generierten Makefiles sind in der Lage, durch die Definition entsprechender Aktionen die relevanten Funktionalitäten der mspgcc-Software aufzurufen. Um nun den Kompilierungsprozess sowie das serielle Flashing innerhalb von Eclipse zu veranlassen, wird das XML-basierte Ant-Plugin eingesetzt. Demzufolge besteht die Herausforderung bei der Integration des Ant-Plugins in der Fähigkeit, auf die Aktionen der Makefiles zuzugreifen und diese aufzurufen, wodurch die mspcc-Software innerhalb des Eclipse-Rahmenwerks gesteuert werden kann.

5.6. ScatterFactory2

Wie bereits geschildert, ist ScatterFactory für die erste Generation der ScatterWeb WSN-Plattform entwickelt worden und somit auf die ESBs (Embedded Sensor Boards) optimiert. Damit die Vorteile von ScatterFactory für die zweite Sensorknotengeneration (MSBs[58]) von ScatterWeb erhalten bleiben, wurde ScatterFactory2 [Kame08] angelehnt an ScatterFactory entworfen und den Anforderungen der MSBs angepasst.[59] Die MSBs unterscheiden sich vom eher monolithischen Vorläufer (ESB) im Wesentlichen durch ihren modularen Entwurf und zwar sowohl softwareseitig als auch hardwareseitig. So verfügt die Softwarearchitektur der MSBs über einen modularen Aufbau der Firmware, bestehend aus diversen Bibliotheken und Trei-

[58] Die Modular Sensor Boards (MSBs): „For any real-world application, a robust hardware platform is required, that allows gearing into all important hardware parameters to achieve optimum results in software applications. For this reason we created the ScatterWeb hardware platform, called the 'embedded sensor board' (ESB) in 2003. The success of the platform led to the spin-off of ScatterWeb GmbH in 2005. Based on the know-how and experiences gained, an industrial platform was designed and certified for radio spectrum and electromagnetic compatibility. Since demands are manifold and rapidly changing, a new research platform, the 'modular sensor board' (MSB) has been created. While still based on an MSP430 series microcontroller, the layout and peripherals have been completely redesigned to better fit research needs of the near future. The new platform is available to the public since the beginning of 2007 through the ScatterWeb GmbH." [Baar07]

[59] Die Anpassungen hinsichtlich der MSB-Eigenschaften zeigen sich deutlich im Metamodell von ScatterFactory2. Eine Darstellung des Metamodells in UML-Notation findet sich in Abbildung 89 im Anhang 11.4.

bern, die dynamisch zur Laufzeit eingebunden werden können. Demzufolge können Firmwaremodule (Infrastrukturcode) für eine Anwendung maßgeschneidert generiert werden. Daher ist es naheliegend, die Laufzeitumgebung einer auf einem MSB laufenden Anwendung als formales Modell zu repräsentieren, durch das eine dynamische Konfiguration der Anwendung mit sämtlichen benötigten Artefakten (Treibern und Bibliotheken) automatisiert bewerkstelligt werden kann. Dies wird mit ScatterFactory2 erreicht.

Abbildung 46 zeigt ein Modell in ScatterFactory2, in dem die Laufzeitumgebung zweier MSB-Anwendungen in einem einzigen Diagramm mitsamt ihrer Konfiguration repräsentiert wird:

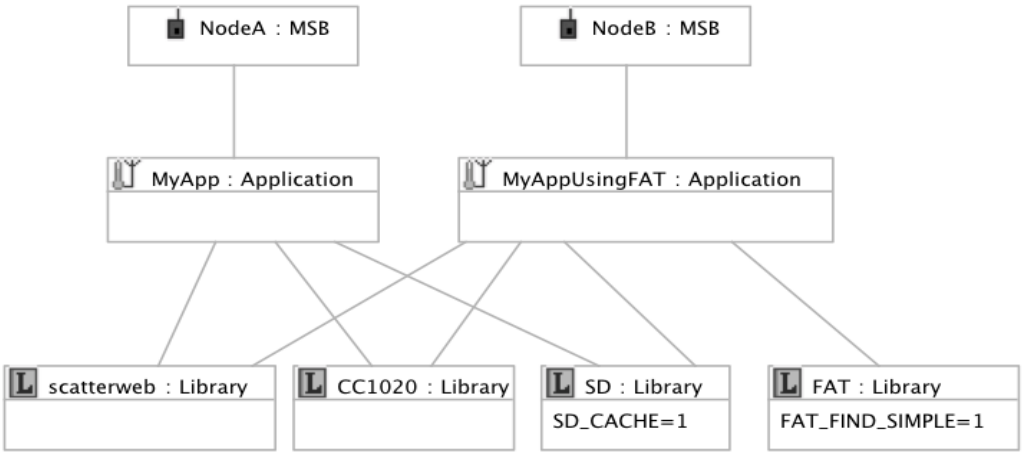


Abbildung 46 Das Modell einer Laufzeitumgebung zweier MSB Anwendungen [Kame08]

Im obigen Modell werden die Anwendungen *MyApp* und *MyAppUsingFAT* repräsentiert. Gemäß der Laufzeitkonfiguration des Modelldiagramms verwenden beide Anwendungen die Standardsystembibliotheken *scatterweb* und *CC1020*. Des Weiteren benutzen beide zusätzlich die SD-Bibliothek, durch die ein Treiber zur Verfügung gestellt wird, um einen Zugriff auf die SD-Speicherkarte des MSBs zu ermöglichen. Das *SD_CACHE* Attribut der SD-Bibliothek wird dabei auf 1 gesetzt. Dies bewirkt die Aktivierung des Caching-Dienstes während der Inanspruchnahme des Moduls der Speicherkarte.

Hinter der Aktivierung der Dienste durch ihre zugehörigen Attribute im Modell verbirgt sich die Idee, dass die Initialisierungsroutine solcher Dienste auf Implementierungsebene als *#define* in der C-Sprache realisiert ist. Ist die Belegung des Attributs im Modell 1 bzw. „wahr“, so wird die Routine des Dienstes aufgerufen.[60] Somit kann man auf einfache Weise beliebige Standarddienste sowie zusätzliche Dienste (Add ons) aktivieren und deaktivieren, wodurch die gesamte Laufzeitumgebung innerhalb des Diagramms sich gemäß den Anforderungen der jeweiligen Anwendung beliebig konfigurieren lässt. Beispielsweise bewirkt die Aktivierung des Cachings einen schnelleren Zugriff auf die in der Speicherkarte abgelegten Daten. Wird der Dienst hingegen vom Benutzer im Modell deaktiviert, so wird dadurch ca. 10% des Speicherplatzes gespart, der sonst für das Caching benötigt würde. Die

[60] Hierzu werden die Werte der Attribute im Modell eingelesen und an die darauf folgende Codegenerierung delegiert, damit die Templates entsprechend eingebunden werden. Im Großen und Ganzen ähnelt dieser Mechanismus den in ScatterFactory eingesetzten Filterregeln.

Wahl der passenden Konfiguration für die zugrunde liegende Anwendung obliegt dem Benutzer, und er kann sie durch einfache Interaktion im Modell treffen.

Ein weiteres im obigen Modell illustriertes Beispiel für die flexible Konfiguration der Laufzeitumgebung mehrerer Anwendungen in einem einzigen Modelldiagramm ist die Konfiguration der Funktionalität des FAT-Filesystems,[61] der Speicherkarte. Hier wird das FAT-Filesystem allerdings lediglich für die Anwendung MyAppUsingFAT aktiviert. Bei der automatisierten Generierung der benötigten Artefakte zur dynamischen Konfiguration der Laufzeitumgebung werden mit Hilfe von ScatterFactory2 diverse Hilfsdateien mitgeneriert, wie XML-Build für das ANT-Plugin sowie die nötigen Makefiles für den Kompilierungsprozess.

In der bisherigen Behandlung des automatisierten Testprozesses in Visual ScatterUnit wurde stets vorausgesetzt, dass die zu testende Anwendung – also das Testobjekt – bereits vorliegt. Daher wurde die Möglichkeit, dass eine Anwendung gleichzeitig entwickelt und getestet wird, bisher nicht behandelt. Eine umfassende Handhabung des Entwicklungsprozesses der Anwendung verlangt jedoch, die Anwendungsentwicklung und den Anwendungstest auch gemeinsam zu betrachten, da dadurch die Effizienz des gesamten Prozesses optimiert wird. Da in ScatterFactory2 die Laufzeitumgebung einer Anwendung für verschiedene Sensoren unterschiedlich konfiguriert werden kann, muss man im Modell eindeutig ausdrücken können, mit welcher Laufzeitkonfiguration der Testfall einer Anwendung assoziiert ist. Es ist beispielsweise möglich, dass nicht alle zu modellierenden Sensoren eines WSNs über die gleiche Hardwareausstattung verfügen, was zur Folge hat, dass ihre Laufzeitumgebung entsprechend unterschiedlich im Diagramm modelliert werden muss. Jedoch unterliegen die Sensorknoten dem gleichen Testszenario und somit dem gleichen Testfall, zu dem sie alle im Modell assoziiert werden müssen. Umgekehrt gilt auch, dass bestimmte Testfälle eine gewisse Konfiguration der Laufzeitumgebung voraussetzen. So kann ein Testfall, der die Simulation von Sensormessungen benötigt, nur mit denjenigen Sensorknoten assoziiert werden, die über die entsprechende Konfiguration dieser Module verfügen.

Kann eine fehlerfreie Verknüpfung zwischen den Testfällen und ihren korrespondierenden Laufzeitkonfigurationen der unterschiedlichen Sensorknoten modelliert werden, so ist die gewünschte Integration des Testprozesses im Kontext der Softwarefamilienfertigung erreicht. Daher wurde die Modellierung in ScatterFactory2 derart erweitert,[62] dass Instanzen, die konkrete Testfälle repräsentieren, mit den in diesen Testfällen involvierten Sensorknoteninstanzen im Modell verbunden werden können, wobei diesen ihre Laufzeitkonfiguration jeweils im selben Modell zugewie-

---

[61] „Das File Allocation Table (FAT; auf Deutsch etwa „Dateizuordnungstabelle“) ist ein Dateisystem und wurde von Microsoft entwickelt. [...] FAT ist die einzige bedeutende Neuerung von QDOS gegenüber CP/M. Nach langen Rechtsstreitigkeiten und einem zweijährigen Patentierungsverfahren wurde am 10. Januar 2006 das Patent und die damit verbundene enorme Marktmacht für FAT (FAT12, -16, -32 etc.) der Herstellerfirma Microsoft zugesprochen. Das deutsche Bundespatentgericht erklärte eines der beiden diesen Schutz begründenden Patente, bzgl. eines gemeinsamen Speicherbereiches für lange und kurze Dateinamen – DE 69429378 bzw. EP 0618540 oder US 5,758,352 – jedoch durch ein Urteil vom 26. Oktober 2006 – Akz. 2 Ni 2/05 (EU) – für nichtig.“ [WebFAT]

[62] Die Erweiterung zeigt sich anhand der Testfall-Instanz im Metamodell von ScatterFactory2 (vgl. Abbildung 89 im Anhang 11.4). Die Testfall-Instanz steht dann für einen konkreten, mit Visual ScatterUnit entwickelten Testfall, der mit der Laufzeitumgebung von ScatterFactory2 verknüpft werden muss, um die Integration beider zu erreichen.

sen wird. Dies zeigt Abbildung 47 eines erweiterten Diagramms von ScatterFactory2, das veranschaulicht, wie die Assoziation zwischen dem Testfall und der Laufzeitumgebung unter Betrachtung der beteiligten Sensorknoten vonstatten geht.

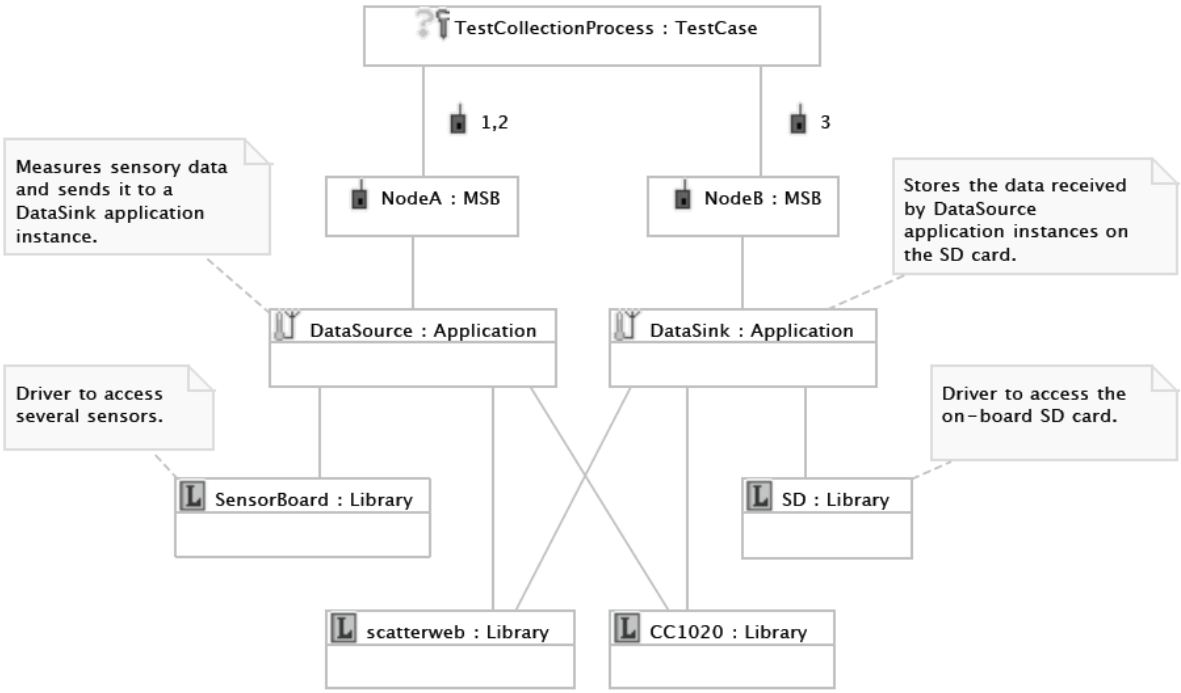


Abbildung 47 Eine Laufzeitumgebung mit einem assoziierten Testfall [Kame08]

Wie aus dem obigen Modell ersichtlich wird, ist der mittels Visual ScatterUnit entwickelte Testfall *TestCollectionProcess* mit den jeweiligen Instanzen der Knotenlaufzeitkonfigurationen NodeA und NodeB verbunden. Diesen Verbindungen werden dann konkrete Sensorknoten anhand ihrer IDs (1, 2 und 3) zugewiesen. Dabei handelt es sich um diejenigen Sensorknoten, die im modellierten Testfall *TestCollectionProcess* vorkommen. Somit können mit Hilfe des obigen Modells sämtliche Codeartefakte automatisiert generiert werden, die für die auf den drei Knoten laufenden Skripte des Testfalls benötigt werden, sowie für die auf den Knoten laufenden Anwendung mitsamt ihrer Treiber und Bibliotheken. Gemäß dem obigen Modell wird die Anwendung *DataSource* für die Knoten 1 und 2 generiert und die Anwendung *DataSink* für den Sensorknoten 3. Hierbei fungieren NodeA und NodeB als Typen, die eine bestimmte Konfiguration einer Anwendungslaufzeitumgebung repräsentieren, denen dann konkrete Sensorknoten zugewiesen werden, für die der Code der Laufzeitumgebung vollständig und automatisiert generiert wird. Das Diagrammmodell des Testfalls *TestCollectionProcess* für die drei Sensorknoten könnte beispielsweise mit Hilfe von Visual ScatterUnit wie in Abbildung 48 dargestellt modelliert werden (vgl. Abbildung 48). Werden die Skripte des obigen Testfalls für die drei Sensorknoten aus dem Gesamtmodell (vgl. Abbildung 47) generiert, so wird für jeden Sensorknoten das zugehörige Makefile mit generiert, um den Compiler entsprechend zu steuern. Dadurch wird gewährleistet, dass die Testskripte eines Sensorknotens mit den Codeartefakten der Anwendungslaufzeitumgebung dieses Sensorknotens zu einem gemeinsamen Code-Image durch den Compiler zusammenge-

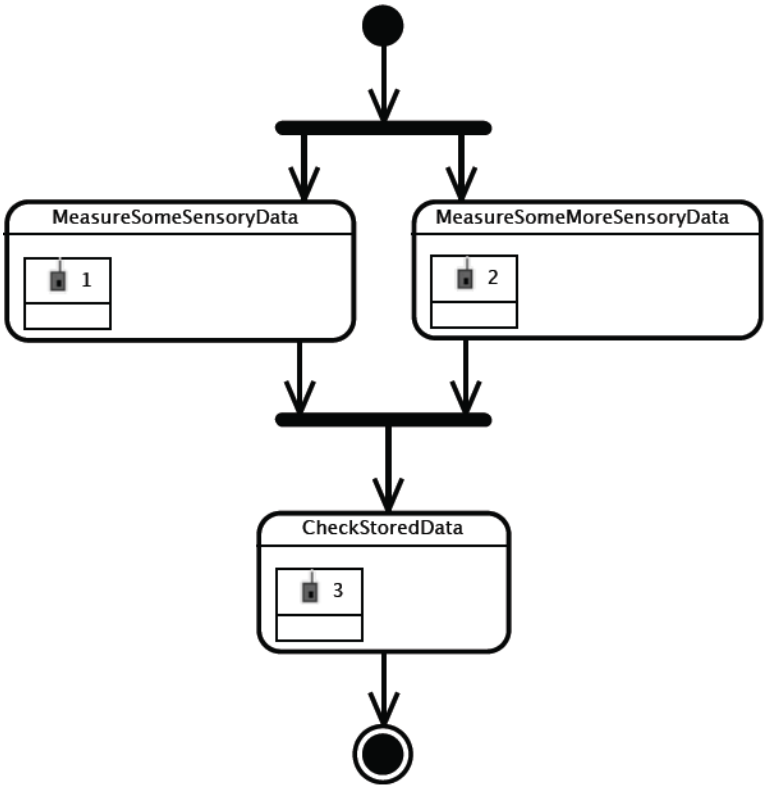


Abbildung 48 Das mit der Laufzeitumgebung assoziierte Diagramm des Testfalls *TestCollectionProcess* [Kame08]

fügt werden. Diese Vorgehensweise der Modellierung einer Assoziation zwischen dem Testfall der zu testenden Anwendung und ihrer Laufzeitumgebungskonfiguration als Softwarefamilie trägt zu einer deutlichen Steigerung der Produktivität des Testprozesses bei. Der Benutzer muss sich nicht mehr darum kümmern, dass die zahlreichen, variierenden Codeartefakte (Testskripte, Treiber, Bibliotheken, etc.) entsprechend zusammengefügt und dem richtigen Zielsensorknoten zugewiesen werden. Die hierzu benötigten Informationen sind alle im Modell erfasst, so dass die korrespondierenden Codeartefakte stets zusammengefügt werden.

5.7. Zusammenfassung

In diesem Kapitel wurde das modellgetriebene Rahmenwerk ScatterFactory behandelt. Es wurde geschildert, wie die auf den Knoten laufenden Anwendungen formal modelliert werden, um aus den Modellen die entsprechenden Softwareartefakte der Anwendung automatisiert zu generieren und anschließend auf die jeweiligen Sensorknoten auszubringen. Zum Zweck der Anwendungsmodellierung wurde eine Domänensprache entwickelt, auf die beispielhaft eingegangen wurde. Hierzu wurde auf Basis von Ecore ein komponentenbasiertes Metamodell entwickelt, mit dessen Hilfe die Semantik und Syntax der Domänensprache zur Anwendungsmodellierung festgelegt wird. Die visuelle modellbasierte Erstellung der Anwendung geschieht mittels des generativen Editors GMF. Für die Überprüfung der Randbedingungen, an die das jeweilige Modell gebunden ist, wurde eine Echtzeit-Validierung in den

Editor auf Basis der Checks-Sprache integriert, um den Entwicklungsprozess robuster zu gestalten.

Des Weiteren wurde ScatterFactory2 vorgestellt, das Nachfolgerahmenwerk, das speziell für die zweite Sensorknotengeneration optimiert wurde. Es wurde ferner geschildert, wie mit Hilfe der modularen formalen Modelle die Laufzeitumgebung der zu entwickelnden Anwendung vollständig und dynamisch konfiguriert werden kann. Alle benötigten Artefakte (Firmwaremodule, Treiber und Bibliotheken) können automatisiert erstellt werden. So dient die automatisierte Laufzeitkonfiguration weiterhin der maßgeschneiderten Codeerzeugung, wodurch nur so viel Code für die Sensorknoten erzeugt wird, wie für ihre jeweilige Rolle erforderlich ist. Dadurch wird nicht nur der knappe Speicherplatz optimiert, sondern es werden unnötige rechen- und energieintensive Softwaremodule vermieden. Der sinkende Anteil an manuellem Code reduziert zugleich die Möglichkeit von Flüchtigkeitsfehlern der Programmierer.

Die größte Leistung von ScatterFactory2 liegt in der Fähigkeit der Integration der Testfallmodelle in die formalen Modelle der Laufzeitumgebung. Somit wird es möglich, den Testcode für das Testobjekt sowie das Testobjekt selbst gemeinsam auf übersichtliche Weise modellierbar zu machen, wodurch der benötigte Code in einem Guss generiert werden kann. Demzufolge ist die Testfallentwicklung nicht isoliert, sondern im Kontext der Anwendungserstellung von Softwarefamilien zu betrachten. Wie dies vonstatten geht, wurde ebenfalls an einem Beispiel erläutert.

6. ScatterPlug: Der Pfad zur WSN-Visualisierung und -Analyse

Mit ScatterPlug [Mysl06] wird ein Plugin-orientiertes Rahmenwerk für die Sensorknoten der WSN ScatterWeb-Plattform auf der Basis von Eclipse vorgestellt. Neben Basisdiensten wie der Konfiguration von Sensorknoten stellt ScatterPlug auch komplexe Dienste zur Verfügung wie die Emulation und Modellierung der Knoten des WSNs im objektorientierten Paradigma, die statistikorientierte Analyse der WSN-Umgebung sowie die Visualisierung der analysierten WSN-Daten und des Status der einzelnen Sensorknoten.

In den folgenden Abschnitten des Kapitels werden die Realisierung und die Merkmale der einzelnen Komponenten von ScatterPlug vorgestellt: die Kommunikationsschicht, die als Brücke zwischen dem Eclipse-Rahmenwerk und der ScatterWeb-Plattform fungiert; das Root-Plugin, das als Grundgerüst der Architektur dient; das Main-View und das Statistic-View, die auf Basis des vorher behandelten Root-Plugins arbeiten, sowie deren Zusammenarbeit.

6.1. Die Kommunikationsschicht

Eine Brücke zur bidirektionalen Kommunikation zwischen dem Eclipse-basierten ScatterPlug-Rahmenwerk einerseits und dem mittels des Rahmenwerks errichteten WSN andererseits ist unerlässlich, um von den Diensten des Rahmenwerks nach der Inbetriebnahme des WSNs problemlos Gebrauch machen zu können. Die Herausforderung – vor allem in Hinsicht auf den erforderlichen Datenaustausch – liegt darin, dass die auf den Sensorknoten laufende Software vollständig in C implementiert ist, während die Sprache des Eclipse-basierten Rahmenwerks Java ist. Hierfür wird die in 3.1 besprochene COM-PORT-Bibliothek von Sun verwendet, auf der die Realisierung der Kommunikationsbrücke fußt. Dabei wird die (serielle) Hardwareschnittstelle zur Kommunikation zwischen dem eGate und dem Rahmenwerk als Java-Objekt dargestellt, dessen Funktionalität mittels entsprechender Methodenaufrufe der Bibliotheks-API gemäß dem objektorientierten Paradigma in Anspruch genommen werden kann.

Da die Kommunikationsschnittstelle zum WSN der objektorientierten Methode folgt, liegt es nahe, das mittels des Rahmenwerks errichtete WSN in der Form korrespondierender Objekte zu emulieren, um von den Vorteilen der objektorientierten Methode sowie den diversen zur Verfügung stehenden objektorientierten Technologien (wie Logging der Nachrichten etc.) Gebrauch machen zu können. Demzufolge wird in der Darstellung des errichteten WSNs ein höheres Abstraktionsniveau erreicht, wodurch die Handhabung sowie Erfassung der Knoten und ihrer Abläufe im Betrieb deutlich erleichtert wird. So kann beispielsweise auf den Status eines Knotens durch Instanzattribute zugegriffen werden, wobei seine Funktionalitäten dann mittels entsprechender Methodenaufrufe in Anspruch genommen werden können.

Zum Zweck der Realisierung der angestrebten Emulation des WSNs wurde das Paket *ScatterWeb for Java* entwickelt, durch das die Einheiten eines WSNs durch ihre jeweils auf dem Rechner laufenden Java-Objekte repräsentiert werden. Nun können netzbezogene Abläufe anhand entsprechender Java-Anwendungen gemäß dem objektorientierten Paradigma realisiert werden. Diese Vorgehensweise ermöglicht die Abstraktion von lästigen technischen hardwarenahen Aspekten. Das Datenmodell des Pakets *ScatterWeb for Java* verfügt über einen baumartigen Aufbau, der sich aus den folgenden grundlegenden Klassen zusammensetzt (vgl. Abbildung 49).

- *Network* fungiert als Behälter, der die (unten aufgelisteten) Instanzen beinhaltet, aus denen das WSN besteht. Desweiteren werden Konfigurationsdaten zum WSN in dieser Klasse abgelegt. Diese Klasse stellt die Wurzel des Datenmodells dar.

- *eGate* repräsentiert die Senke. Einer *Network*-Instanz ist genau eine Senke bzw. *eGate*-Instanz zugewiesen.
- *Node* repräsentiert einen Sensorknoten. Eine *Network*-Instanz kann beliebig viele Node-Instanzen enthalten.
- *AbstractModule* repräsentiert ein auf dem Sensorknoten laufendes Softwaremodul. Ein Knoten besteht aus einer beliebigen Anzahl solcher Module, die gemeinsam die Sensormaske bilden.

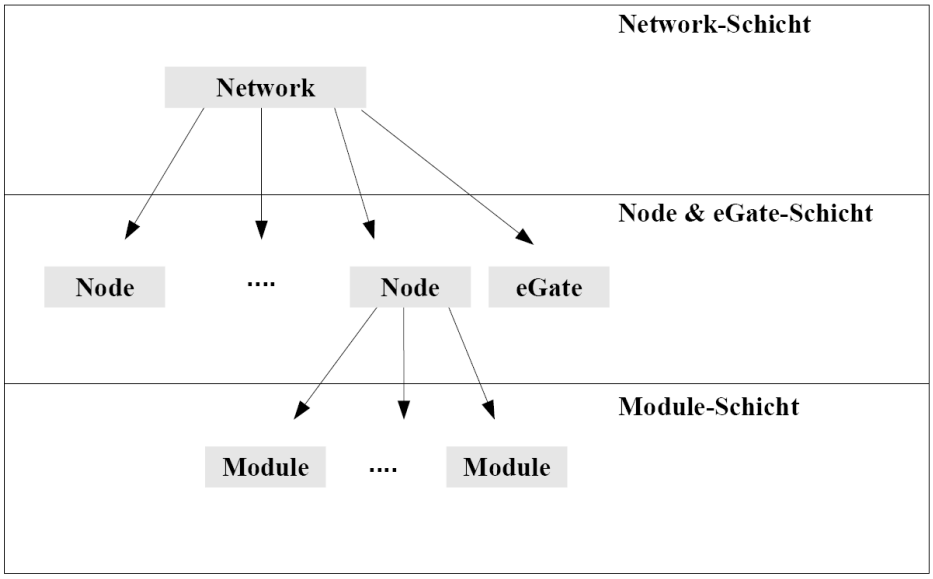


Abbildung 49 Das Datenmodell in ScatterWeb for Java [Mysl06]

Abbildung 50 veranschaulicht die Ablaufschritte beim Nachrichtenaustausch und die daran beteiligten Klassen bzw. Methoden des ScatterWeb for Java-Pakets. Ausschlaggebend für den Nachrichtenaustausch in ScatterWeb for Java ist, dass die Zustellung der aus dem WSN kommenden Nachrichten einem ereignisbasierten Ansatz unterliegt, der auf der Java-Listener-Technik basiert. Dies ist ein weiteres Beispiel für den Einsatz objektorientierter Technologien.

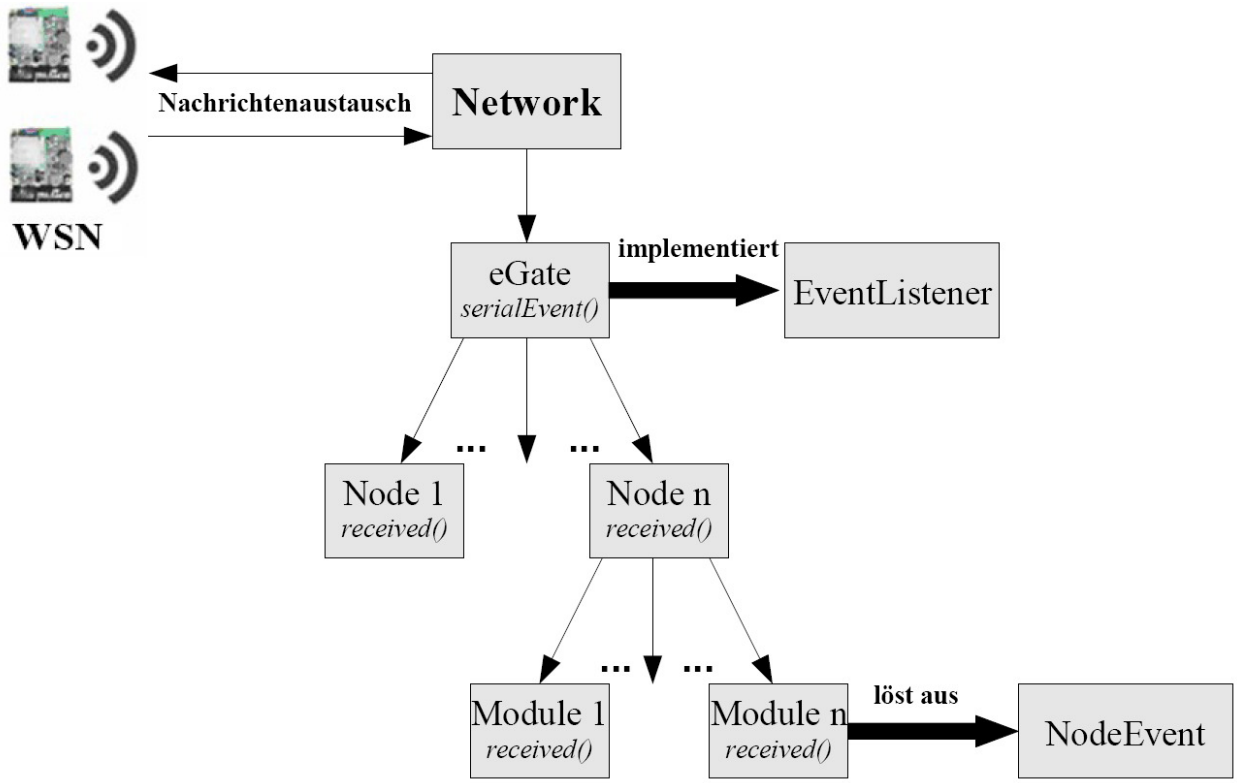


Abbildung 50 Nachrichtenaustausch in ScatterWeb for Java

Die eGate-Klasse implementiert die von der COM-Port-API angebotene Schnittstelle `javax.comm.SerialPortEvent.Listener`. Dementsprechend wird die lauschende eGate-Methode `serialEvent()` aufgerufen, sobald eine Nachricht aus dem WSN vorliegt. Die Methode bewirkt die Zustellung der Nachricht an denjenigen Knoten, an den die Nachricht adressiert ist. Der Empfang der Nachricht wird durch die Methode `received()` bewerkstelligt. Gemäß dem modularen[1] Aufbau der Node-Klasse korrespondiert jede empfangene Nachricht mit einem in der Node-Klasse vorhandenen Modul, durch das die Nachricht bearbeitet wird. Eine solche Nachricht bewirkt in der Regel die Änderung der Attribute eines Moduls. Infolgedessen feuert das betroffene Modul ein Ereignis vom Typ `NodeEvent`. Jedes Objekt bzw. jede Schnittstelle in der Emulationsinfrastruktur, die dieses Ereignis implementiert, wird nun über die sich eben ereignete Wirkung der Nachricht benachrichtigt und kann entsprechend darauf reagieren.

Ein weiterer hilfreicher Aspekt beim Nachrichtenaustausch in *ScatterWeb for Java* ist die Fähigkeit, den Nachrichtenverkehr zu protokollieren (loggen). Hierzu wurden diverse, für diesen Zweck hilfreiche Java-basierte Technologien eingesetzt. Es handelt sich hier um zwei Kategorien von Daten, die protokolliert werden: Einerseits muss der gesamte Datenverkehr der vom Netz versendeten bzw. empfangenen Nachrichten protokolliert werden, andererseits muss der sich durch die Wirkung dieses Nachrichtenaustausches ständig ändernde Status des Sensornetzes mit sämtlichen Entitäten (eGate, Knoten, usw.) protokolliert werden. Für ersteres wird Log4j [WebLog4j] von Apache verwendet, wodurch die zu protokollierenden Nachrichten nach einem zuvor vom Benutzer in einer zentralen Datei festgelegten Format abgelegt werden, die in der Log4j mit *Properties* bezeichnet wird. Abbildung 51 veranschaulicht diesen Vorgang:

[1] Die Strukturierung der Node-Klasse in vom Benutzer beliebig konfigurierbare Module entspricht dem modularen Aufbau auf Hardwareebene bezüglich der neuen Knoten Generation MSB (Modular Sensor Board). Dieser Aspekt war ausschlaggebend für die Entwurfsentscheidung.

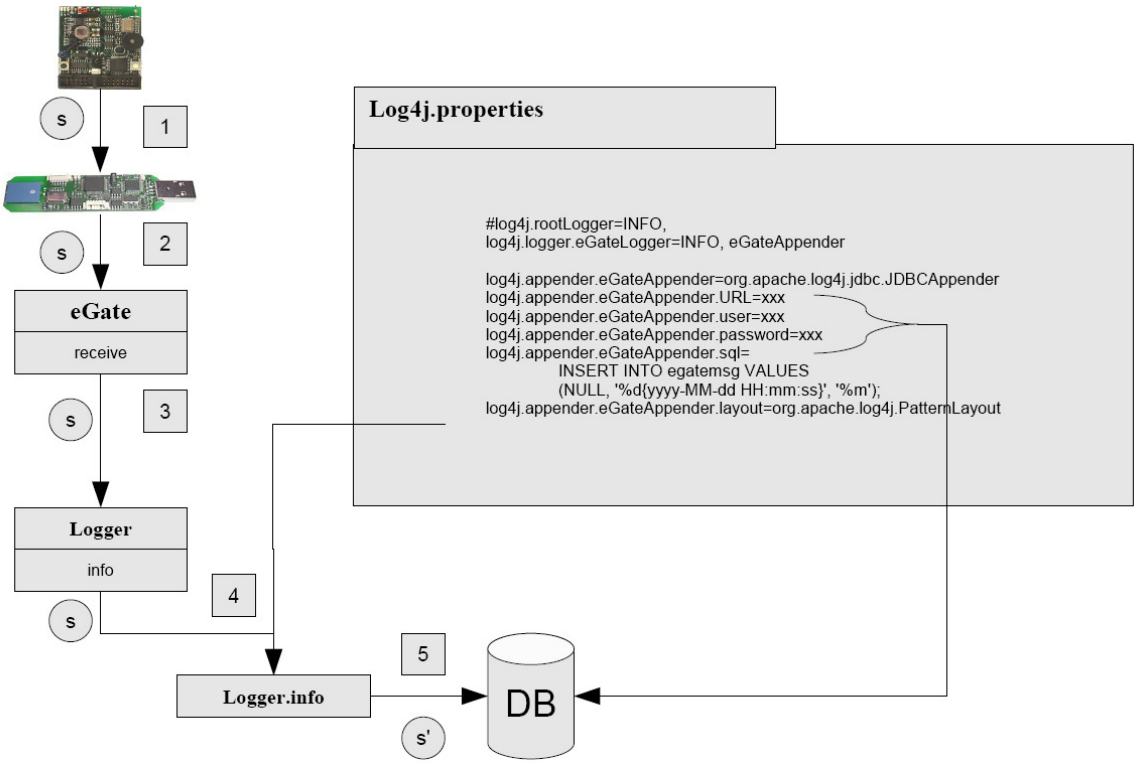


Abbildung 51 Einsatz von Log4j zur Protokollierung der Nachrichten [Mysl06]

Zuerst wird die vom WSN stammende Nachricht s (siehe 1) via eGate (2) mit Hilfe ihrer receive() Methode (3) empfangen und anschließend für das Logger Modul aufbereitet. Das Logger Modul transformiert die empfangene Nachricht gemäß dem vom Benutzer individuell definierten Muster in der Properties Datei zu s´ (4), diese wird abschließend mittels der Logger.info in der Datenbank gespeichert. Auf eine Änderung des Nachrichtenprotokolls im WSN kann zu jeder Zeit entsprechend reagiert werden, indem die erforderliche Anpassung des Musters der betroffenen Felder in der Properties-Datei vollzogen wird. Diese muss der Benutzer nur ein einziges Mal vornehmen.

Die Speicherung der WSN-Konfiguration, wie z.B. die Formatierung und Skalierung der Grafik des Feldes, in dem die Knoten platziert werden, die Positionierung der Knoten im Feld[2] sowie der Status jedes einzelnen Sensorknotens, werden mit Hilfe des Castor-Rahmenwerks realisiert (vgl. Abbildung 52). Dabei werden, wie die folgende Abbildung veranschaulicht, sämtliche als Objekte vorliegende Daten durch das Castor-Rahmenwerk [WebCastor] nach dem XML-Format bzw. dem Schema rationaler Datenbanken automatisiert abgebildet, damit diese für weitere Verarbeitungsschritte durch andere Programme z.B. andere Plugins zur Verfügung stehen. Somit findet eine automatisierte und den Bedürfnissen des Benutzers angepasste Serialisierung der WSN-Daten statt. Dies stellt einen weiteren Vorteil der Emulation des WSNs im objektorientierten Paradigma dar, da solche Technologien optimal auf Objektinstanzen operieren.

[2] Diese sind einige Funktionalitäten des Root-Plugins, die ausführlicher im kommenden Abschnitt behandelt werden.

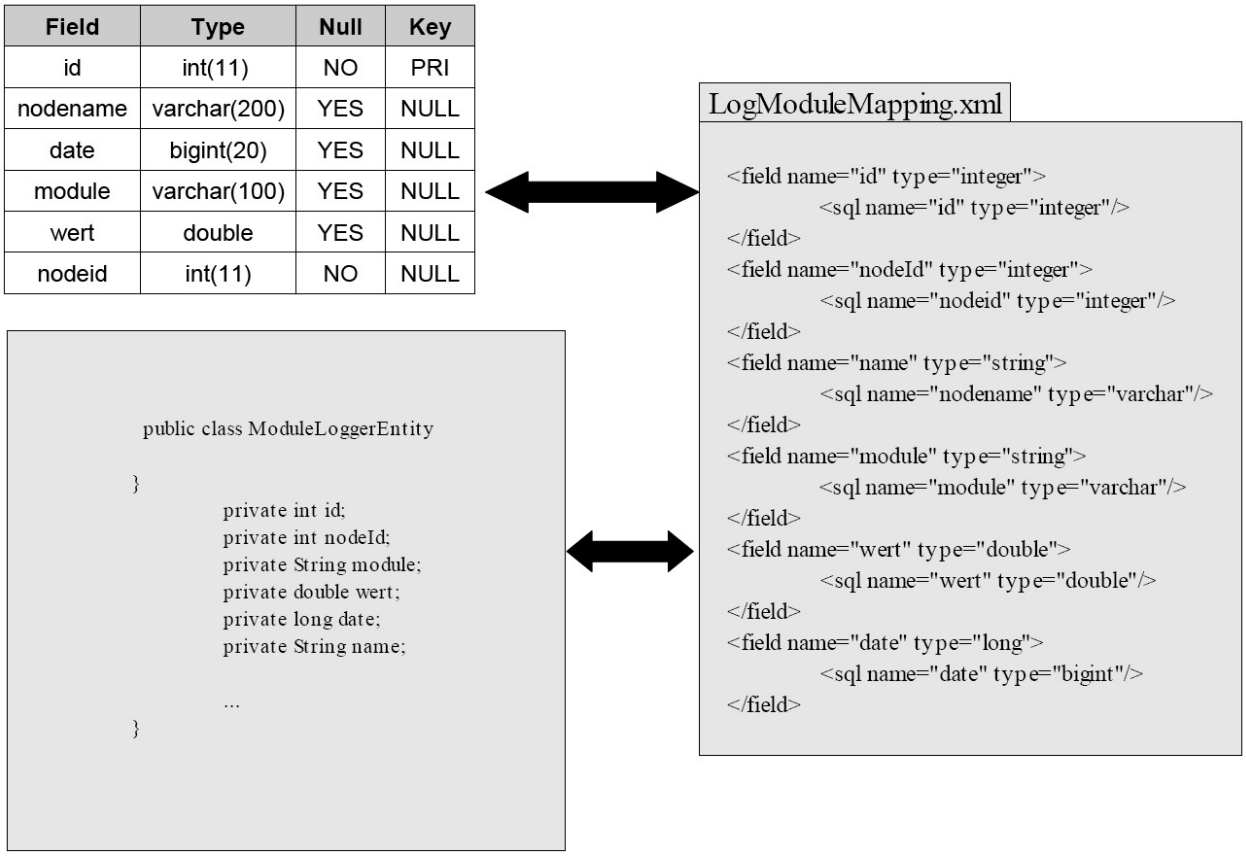


Abbildung 52 Automatisierte Datenserialisierung mit Hilfe des Castor-Rahmenwerks [Mysl06]

6.2. Das Root Plugin

Das Root-Plugin ist das Herzstück des gesamten ScatterPlug-Rahmenwerks. Aus der Sicht des Benutzers ist das Root-Plugin der Ausgangspunkt, um das WSN Eclipse-integriert zu planen, einzurichten und zu konfigurieren. Mit Hilfe des Root-Plugins kann die Grafik des WSN-Grundrisses definiert und gemäß der Entfernungen in der Realität (zweidimensionaler Raum) individuell skaliert und angepasst werden. Im Folgenden werden mittels des Plugins die Sensorknoten in der Grafik an den ihrer realen Positionen entsprechenden Koordinaten vom Benutzer platziert. Hierfür stellt das Plugin die entsprechenden Wizards und Funktionalitäten zur Verfügung, die im Anhang 11.3.1 veranschaulicht werden. Sämtliche aus dem Wizard entstandenen Konfigurationsdateien (z.B. für Castor und Log4j) werden nun automatisch im Verzeichnis des WSN-Projekts in der Eclipse-Umgebung abgelegt (vgl. Abbildung 53). So kann der Benutzer mehrere Sensornetze mit unterschiedlichen

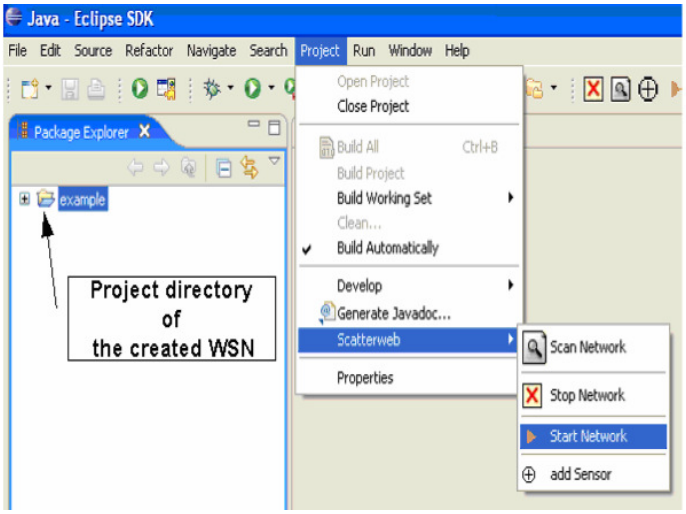


Abbildung 53 Eclipse Projekt für das ScatterWeb WSN [Mysl06]

Konfigurationen kreieren und übersichtlich verwalten. Desweiteren kann der Benutzer mittels des Root-Plugins eine Verbindung zu jedem beliebigen WSN aufbauen und das WSN nach neu hinzu kommenden bzw. nicht mehr vorhandenen Knoten scannen. Neue Knoten werden flexibel vom Benutzer ins WSN aufgenommen:

Die gesamte Konfiguration des WSNs, wie die Daten des Grundrisses (Pfad, Größe, die Koordinaten der vom Benutzer platzierten Sensorknoten sowie der Status jedes einzelnen Knotens), können nun mit Hilfe des CastorRahmenwerks im XML Format abgelegt werden. Die XML-Serialisierung fungiert somit als Austauschmechanismus zwischen den verschiedenen Plugins der gesamten ScatterClipse-Werkzeugkette (vgl. Abbildung 54). Folglich kann jede beliebige Komponente der Werkzeugkette gemäß ihrer Funktionalität das WSN bearbeiten und dann im XML-Format ablegen (exportieren), so dass eine weitere Komponente das WSN im Folgenden importieren, bearbeiten und im XML-Format wiederum exportieren kann, usw. Dadurch wird auf robuste Art und Weise u.a. bewirkt, dass eine beliebige Synchronisation der Konfiguration des WSNs aufrecht erhalten bleibt, was den Grad der Zusammenarbeit zwischen den Plugins der Werkzeugkette erheblich steigert.

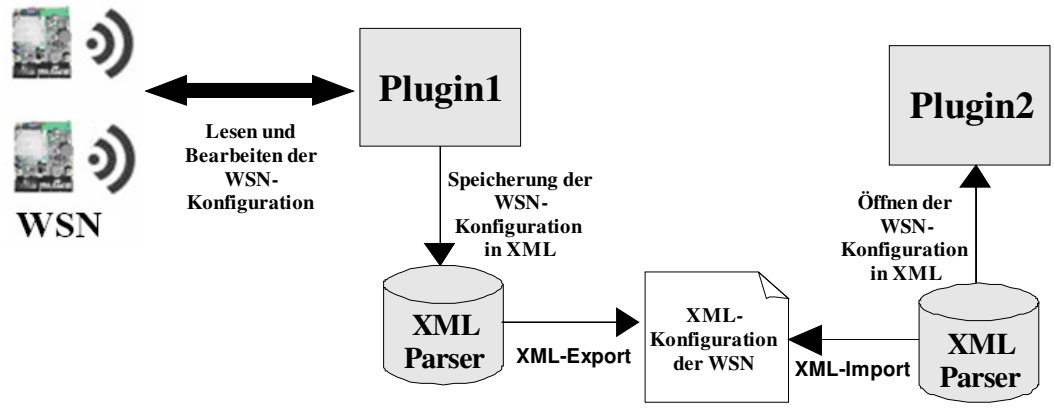


Abbildung 54 XML als Austauschformat in der Plugin-orientierten Architektur [Ding07]

6.3. Main View

Das Main View stellt die Repräsentationsschicht des Rahmenwerks dar, in der die Interaktion mit dem mittels des Root-Plugins errichteten WSN stattfindet. Das Main View ermöglicht es dem Benutzer, die Sensorknoten zu kreieren, zu platzieren und wieder aus dem WSN zu entfernen. Ferner kann der Benutzer mit Hilfe des Main Views die Maske der einzelnen Sensorknoten im Grundriss visuell konfigurieren, indem er ihre jeweiligen Module gemäß ihrer Hardwareausstattung individuell festlegt. Dies veranschaulicht Abbildung 55:

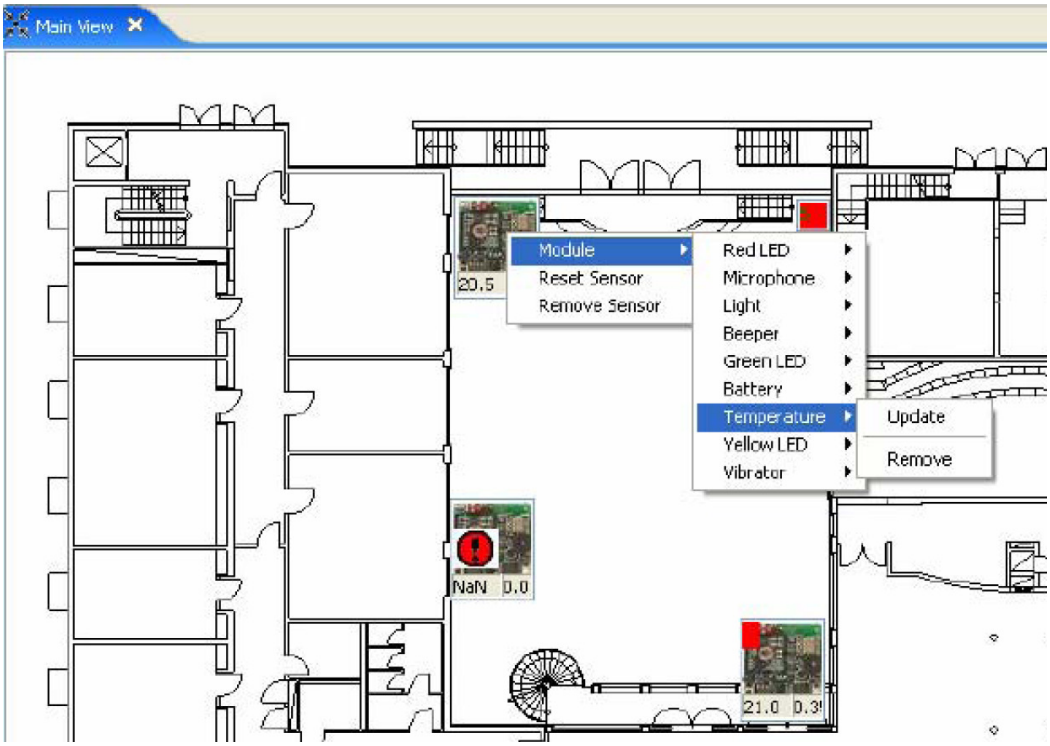


Abbildung 55 Main View [Mysl06]

Des Weiteren werden sämtliche, aus den im WSN aktiven Sensorknoten ankommenden Nachrichten in der jeweiligen Knotendarstellung des Main Views visualisiert, so dass der Benutzer über jede Statusänderung der Knoten und deren Module direkt informiert wird. Eine weitere Aufgabe des Main Views ist die Visualisierung des Status der einzelnen Module. Abbildung 56 zeigt verschiedene Varianten der Visualisierung der Sensorknoten sowie deren Module: Links im Bild wird angezeigt wie die ID des Sensorknotens dem Benutzer angezeigt wird, sobald er mit der Maus über die entsprechende Sensordarstellung fährt. Im mittleren Bild wird die Visualisierung von Modulen illustriert, sobald diese im realen Netz einen Wert messen oder ein Ereignis aufspüren. Im rechten Bild wird die Möglichkeit der punktorientierten Darstellung der Knoten im Main View gezeigt: Die gewählten Knoten können lediglich als Punkte repräsentiert werden, ohne dass deren Module visualisiert werden müssen.[3]

[3] Werden bestimmte Knoten nach dem Scannen des Netzes nicht erreicht, so werden diese als rote Punkte angezeigt.

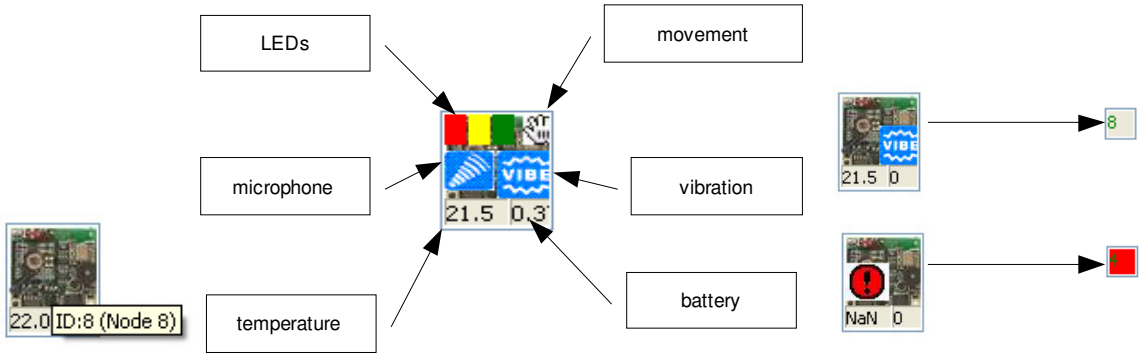


Abbildung 56 Visualisierung der Sensorknoten im Main View [Mysl06]

Dies ist optimal, wenn das WSN aus einer großen Anzahl von Knoten besteht und der Benutzer die Visualisierung aller Sensoren nicht überblicken kann. So wird es dem Benutzer ermöglicht, die für ihn nicht relevanten Knoten punktorientiert darstellen zu lassen und so seinen Fokus auf diejenigen Knoten zu legen, die seine momentane Fragestellung betreffen. Die punktorientierte Darstellung optimiert auf diese Weise die Interaktion mit dem Benutzer und trägt durch die selektive Visualisierung wesentlich zu einer höheren Effizienz bei der rechenaufwendigen grafischen Darstellung der Knoten bei.

6.4. Das Statistik View

Mit Hilfe des Statistik Views können die von den Sensorknoten erfassten Daten statistisch analysiert und entsprechend mathematisch visualisiert werden. In bestimmten Szenarien ist es sehr hilfreich, die von einem Sensor erfassten und bearbeiteten Daten statistisch zu analysieren, um Rückschlüsse auf die Messgenauigkeit der Sensorik der einzelnen Sensorknoten ziehen zu können. Dem gemäß kann ein Ausreißer in einer Serie aufeinander folgender Messwerte eines Sensorknotens ein Hinweis darauf sein, dass die Sensorik des Knotens zum betroffenen Zeitpunkt fehlerhaft gemessen hat. Die Häufigkeit des Auftretens solcher Ausreißer fungiert somit als Indikator für die Qualität der Messung durch die jeweilige Sensorik. Des Weiteren können die Gegebenheiten der von den Sensorknoten zu beobachtenden Umgebung durch statistische Größen präziser erforscht und dargestellt werden. Ferner lassen sich dadurch bestimmte Beziehungen zwischen den sich in der Sensorumgebung fortlaufend ereignenden Abläufen erkennen. So können Korrelationsbeziehungen zwischen verschiedenen erfassten Daten aufgedeckt und angegeben werden, wie die Beziehung zwischen der Lichtstärke und der Temperatur in einem Gebiet. Desweiteren kann der Einfluss des Energieverbrauchs (Batterie) von Modulen angegeben werden.

Zu diesem Zweck muss der Benutzer in die Lage versetzt werden, auf Werkzeugebene sämtliche Knoten aus dem Main View zu selektieren und beliebige statistische Operationen[4] aus den von diesen Sensorknoten gemessenen Daten zu berechnen (vgl. Abbildung 57). Dies setzt eine Zusammenarbeit zwischen dem Main View und dem Statistik View voraus. Das Statistik View muss es dem Benutzer ermöglichen, zwischen den verschieden statistischen Operationen zu wählen und diese bezüglich einer von ihm festgelegten Zeitspanne auf eine Messserie anzuwenden. Da es von großem Interesse ist, die Messwerte der unterschiedlichen Sensorknoten statistisch zu vergleichen und zu untersuchen, können unterschiedliche Messserien im Statistik View zu einer einzigen

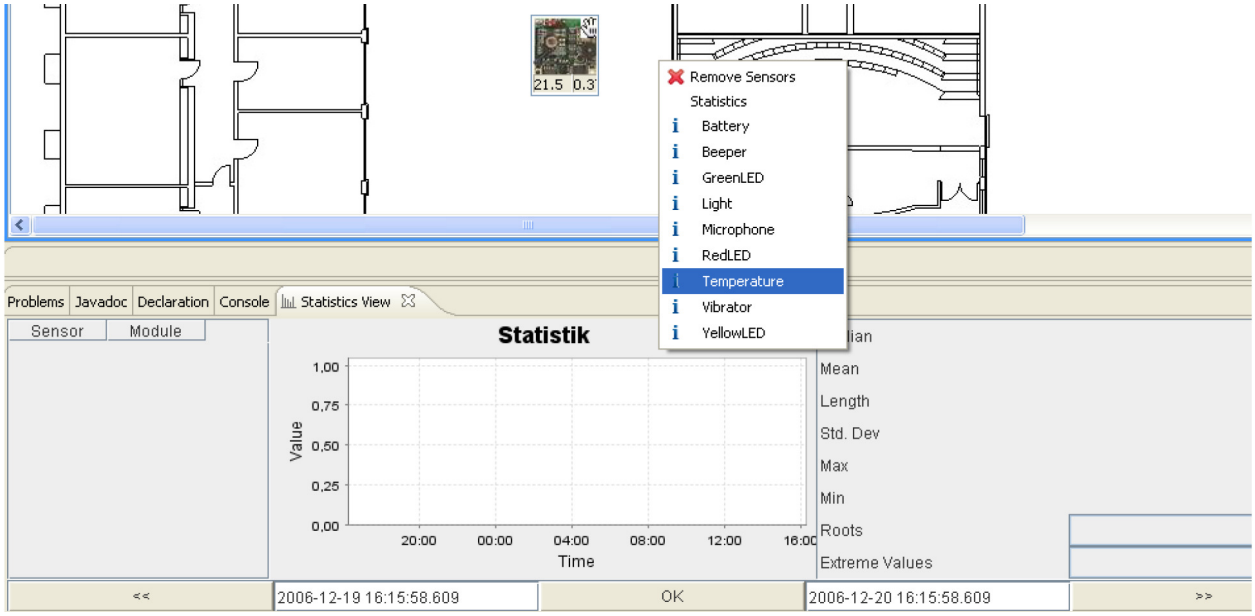
[4] Für die Berechnung der mathematischen Operationen wurde das Paket ScatterWeb.math implementiert, welches zahlreiche statistische Operationen und deren grafische Darstellung zur Verfügung stellt: Diese reichen von einfachen Operation wie der Ermittlung des Medians und extremer Werte bis zur Ermittlung von Ausreißern, Korrelation, Regression, Trendlinien, Standardabweichung und Interpolation, um bestimmte Statistiken als glatte Kurve darstellen zu können, etc.

Serie mathematisch gruppiert werden. Dadurch ist es beispielsweise möglich, eine komplexe Messserie durch die Gruppierung mehrerer Messserien (z.B. zur Temperatur) zu bilden, die von unterschiedlichen Sensorknoten in unterschiedlichen Zeitspannen gemessen wurden. Die Knoten können dabei in verschiedenen Regionen platziert sein. Die Streuung des Ortes und der Zeit wird bei der Berechnung der Statistik in Betracht gezogen. Hinzugefügt werden können Messserien, die *on the fly* ermittelt werden, während die statistischen Operationen und deren zugehörige Darstellung parallel darauf angewendet werden. Folglich kann der Benutzer auch gegenwärtig im WSN ablaufende Ereignisse statistisch untersuchen und darstellen lassen.

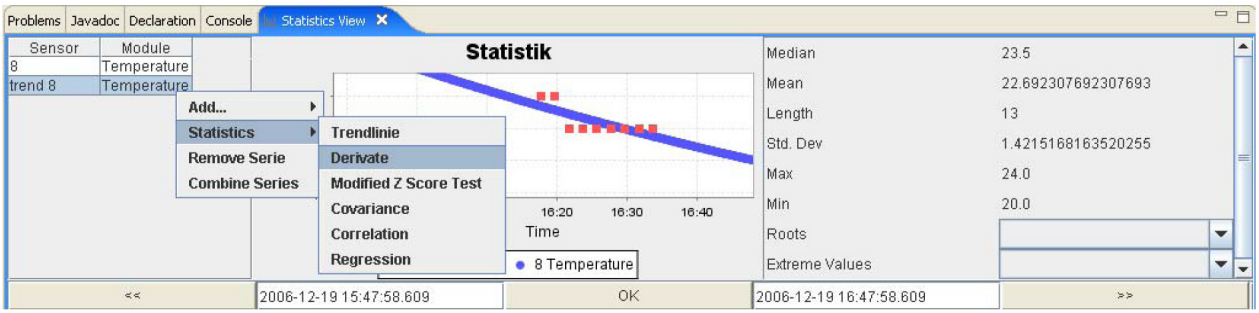
Die Implementierung des Statistik Views beruht nicht nur auf der Eclipse-Grafikbibliothek SWT, (vgl. 2.1) sondern auch auf der Swing-Bibliothek von Java. Dies geht auf die Verwendung des JfreeChart-Werkzeugs [WebJFreeChart] zur mathematischen Visualisierung der Statistikgrafiken zurück. JFreeChart ist ein Swing-basiertes Opensource Werkzeug, das im Vergleich zu anderen verwandten Opensource Werkzeugen eine starke Stabilität im Einsatz aufweist. Ausschlaggebend für den Erfolg der Integration von JFreeChart in das Statistik View ist die Kommunikation zwischen beiden Seiten. Abbildung 58 zeigt, wie diese vonstatten geht sowie die daran beteiligten Komponenten: Wird beispielsweise die Temperatur innerhalb einer bestimmten Zeitspanne untersucht, so wird eine Schnittstelle vom Typ ModuleHistory erzeugt, deren Aufgabe es ist, die notwendigen Daten aus der Datenbank bereitzustellen (Schritt 4 im Bild). Sobald jedoch das angegebene Zeitfenster die aktuelle Zeit überschreitet, werden alle vom WSN via eGate (Schritt 1) ankommenden relevanten Daten direkt (on the fly) selektiert und zur Visualisierung verarbeitet. Hierfür wird mit Hilfe der ModuleHistory eine Nodelistener-Instanz erzeugt,[5] die gemäß der Benutzeranfrage die relevanten ankommenden Nachrichten herausfiltert (Schritt 2) und anschließend für die folgenden Schritte als verarbeitbare Messserien zur Verfügung stellt (Schritt 3). Die Messserien werden durch die Serie Klasse in der Datenstruktur repräsentiert. Die Klasse Serie bildet die kanonische mathematisch zu verarbeitende Entität.

Die ModuleHistory stellt die jeweilige Serie für das entsprechende Modul des ScatterWeb.math Pakets zur statistischen Berechnung bereit (Schritt 5). Im Folgenden wird die aus dem vorausgegangenen Berechnungsschritt resultierende bzw. bearbeitbare Serie visualisiert. Da jedoch die Objekte der Serie Klasse aus Kompatibilitätsgründen nicht durch JFreeChart direkt geplottet werden können, wurde dieses Hindernis mittels der Implementierung der Schnittstelle (TimeSerieEx) bewältigt (Schritt 6). Diese erbt von der Klasse TimeSerie, die von der JfreeChart mitgeliefert und somit visualisiert werden kann (Schritt 7).

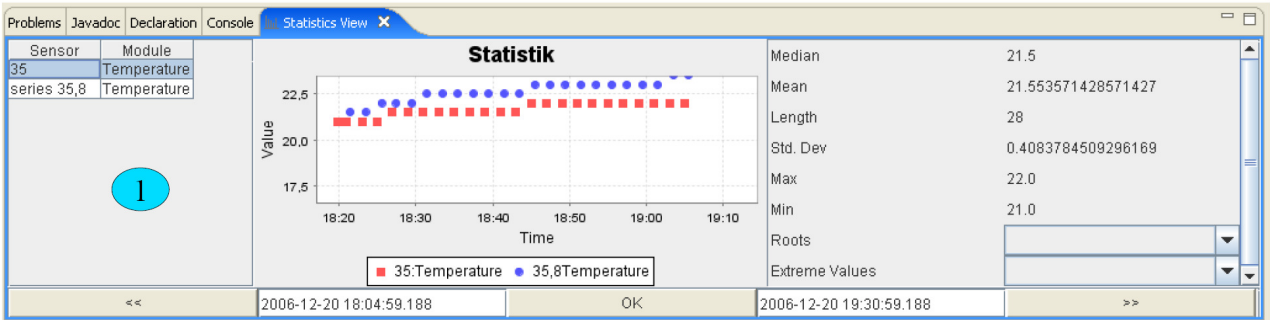
[5] Die Funktionalität der NodeListener-Klasse wurde in 6.1 bei der Behandlung der Kommunikationsschicht erörtert.



a) Auswahl des Sensorknoten und des Typs des zu messenden Wertes (z.B. Temperatur)



b) Wahl beliebiger statistischer Operationen sowie deren Ermittlung bzw. Visualisierung im GUI



c) Gruppierung mehrerer statistischen Messserien (siehe 1)

Abbildung 57 Merkmale des Statistik Views [Mysl06]

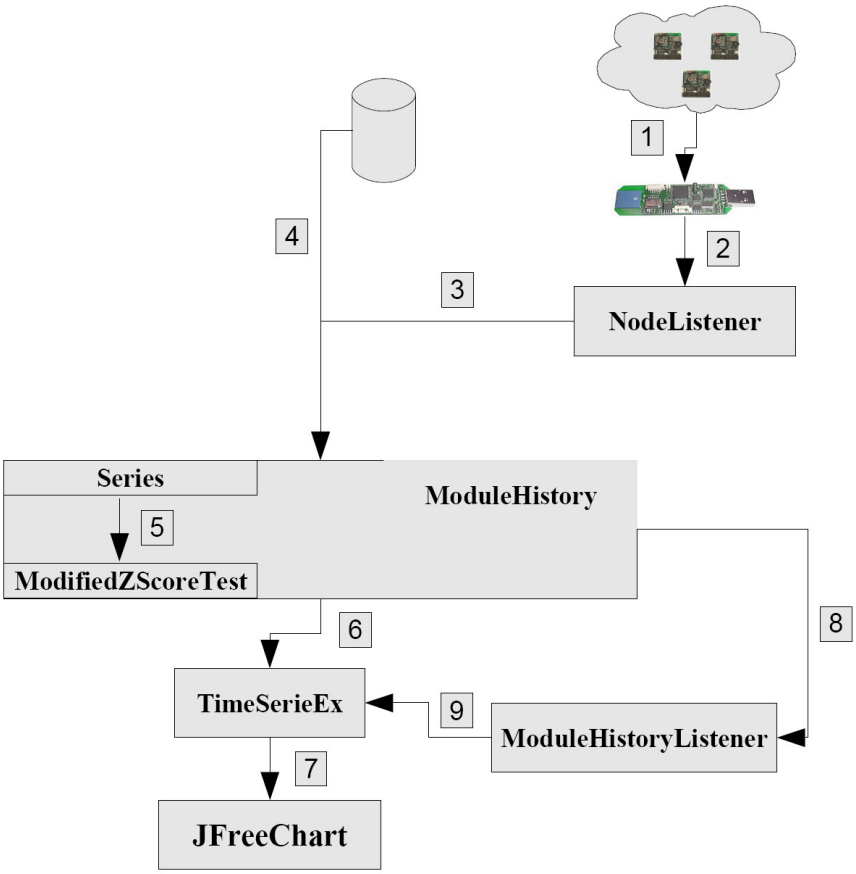


Abbildung 58 Kommunikation zwischen Statistik View und JfreeChart [Mysl06]

6.5. Zusammenfassung

Neben Basisdiensten wie der Konfiguration von Sensorknoten stellt ScatterPlug auch komplexe Dienste zur Verfügung wie die Emulation und Modellierung der Knoten des WSNs im objektorientierten Paradigma, die statistische Analyse der WSN-Umgebung sowie die Visualisierung der analysierten WSN-Daten und des Status der einzelnen Sensorknoten. Desweiteren bietet ScatterPlug die Möglichkeit, WSNs mit Hilfe des Eclipse-Rahmenwerks zu planen und einzurichten, um eine optimale Inbetriebnahme der Sensoren zu erreichen.

Zum Zweck der Realisierung der angestrebten Emulation des WSNs wurde das Paket *ScatterWeb for Java* entwickelt, durch das die Einheiten eines WSNs durch ihre jeweils auf dem Rechner laufenden Java-Objekte repräsentiert werden. Ferner wurde für die Berechnung der mathematischen Operationen das Paket ScatterWeb.math implementiert, welches etliche statische Operationen und deren grafische Darstellung zur Verfügung stellt. Die statistische Analyse der von einem Sensor erfassten und bearbeiteten Daten kann sehr hilfreich sein, um Rückschlüsse auf die Messgenauigkeit der Sensorik der einzelnen Sensorknoten ziehen zu können.

Ein weiteres wichtiges Merkmal von ScatterPlug ist die Fähigkeit, die gesamte Konfiguration des erstellten drahtlosen Netzes im XML-Format zu speichern. Somit können die verschiedenen Plugins die XML-Daten untereinander importieren bzw. exportieren. Folglich fungiert die XML-Serialisierung als Austauschmechanismus zwischen den verschiedenen Plugins der gesamten ScatterClipse-Werkzeugkette.

7. ScatterEditor: Der Pfad zur WSN Konfiguration, Steuerung und Überwachung

Neben Texteditoren zum Bearbeiten von Texten existieren mittlerweile diverse andere Editoren z.B. zur Bearbeitung von Audio- und Grafikdateien sowie von Webseiten. Je komplexer das zu bearbeitende Element ist, desto höher sind die Anforderungen an den jeweiligen Editor. Für gute Editoren gilt, dass sie dem Benutzer die Arbeit erleichtern.

Dieses Prinzip lässt sich auf WSNs übertragen, die in der gegenwärtigen Ära des *ubiquitous and pervasive computing* eine zentrale Rolle spielen und immer mehr Anwendungsbereiche finden. Die Herausforderungen hinsichtlich adäquater Steuerung und Überwachung der sich im Betrieb befindenden Sensorknoten, sowie der Fehlerbehebung steigen entsprechend und damit die Anforderungen an die dazugehörigen Werkzeuge.

ScatterEditor [Ding07] ist ein solches Werkzeug, das die genannten Dienste auf eine illustrative Weise zur Verfügung stellt und so den Benutzer in die Lage versetzt, sie interaktiv in Anspruch zu nehmen und damit das WSN zu „editieren". ScatterEditor ist ein Eclipse-basiertes Rahmenwerk mit einer Plugin-orientierten Architektur. Durch die Kombination der Plugins ermöglicht ScatterEditor die *Over The Air* Konfigurierung, Steuerung und Überwachung der Sensorknoten bereits während ihres Betriebes.

Diese Merkmale können dank der Internet-Integration von ScatterEditor sowohl im lokalen Modus als auch entfernt im Internet-basierten (*remote*) Modus genutzt werden. Neben der Integration von IP und ScatterWeb wird bei der Gestaltung von Scatter-Editor besonderer Wert auf den Aspekt der *Human-Computer Interfaces* für WSN gelegt. Im Folgenden wird die Architektur von ScatterEditor und dessen Komponenten dargestellt. Danach werden die einzelnen Funktionalitäten des Rahmenwerks behandelt sowie deren Internet-Integration.

7.1. Architektur und Design von ScatterEditor

Mit Hilfe des ScatterEditor-Plugins können die Sensorknoten sowie die Senke (eGate) der ScatterWeb-Plattform kontrolliert, konfiguriert und überwacht werden. Wie eingangs erwähnt, kann ScatterEditor dank der Internetintegration sowohl im lokalen als auch im entfernten (*remote*) Modus benutzt werden. Im ersten Fall fungiert das Programm als Server, an dem das eGate direkt angeschlossen wird. Im zweiten läuft das Programm lediglich als Klient und verbindet sich mit dem WSN durch einen designierten Server über das Internet. Dieser Aufbau macht es möglich, dass ScatterEditor je nach Wahl des Benutzers als Klient oder als Server fungieren kann und hat den Vorteil, dass der Benutzer sich nur mit einem Programm vertraut machen muss, was die Flexibilität erhöht. Dies erleichtert den Einstieg und die Akzeptanz von ScatterEditor.

Die zahlreichen Dienste von ScatterEditor wurden durch einzelne Reiter realisiert, zwischen denen der Benutzer leicht navigieren kann. Alternativ zum eGate – als Verbindungsglied zum drahtlosen Netz – ist es ebenfalls möglich, einen gesonderten Knoten, genannt *SuperNode*, einzusetzen. Dieser SuperNode ist ein Sensorknoten mit spezieller Software, die die Funktionalitäten des EGates unterstützt und durch ein serielles Kabel mit dem Rechner verbunden ist. Damit kann der SuperNode alternativ zum EGate die Rolle der Senke problemlos einnehmen.

Diese Entwurfsentscheidung ermöglicht es, dass ScatterEditor hinsichtlich der Kommunikation mit dem drahtlosen Netz sowohl über die serielle als auch die parallele Schnittstelle betrieben werden kann. Dies ist ausschlaggebend, wenn man bedenkt, dass oft andere Werkzeuge bzw. Plugins der ScatterClipse-Werkzeugkette nebenläufig mit demselben WSN kommunizieren. So können die unterschiedlichen Werkzeuge und somit deren Funktionalitäten gleichzeitig in Anspruch genommen werden, wobei eine

Lastbalancierung und eine Trennung von Belangen (*separation of concerns*) bezüglich der Kommunikationsschnittstelle zwischen den Sensorknoten auf der einen und den auf dem Rechner laufenden Komponenten der Werkzeugkette auf der anderen Seite stattfinden. Abbildung 59 veranschaulicht den Aufbau von ScatterEditor bezüglich der Kommunikationsschicht zwischen dem WSN und den Komponenten des ScatterEditor-Plugins.

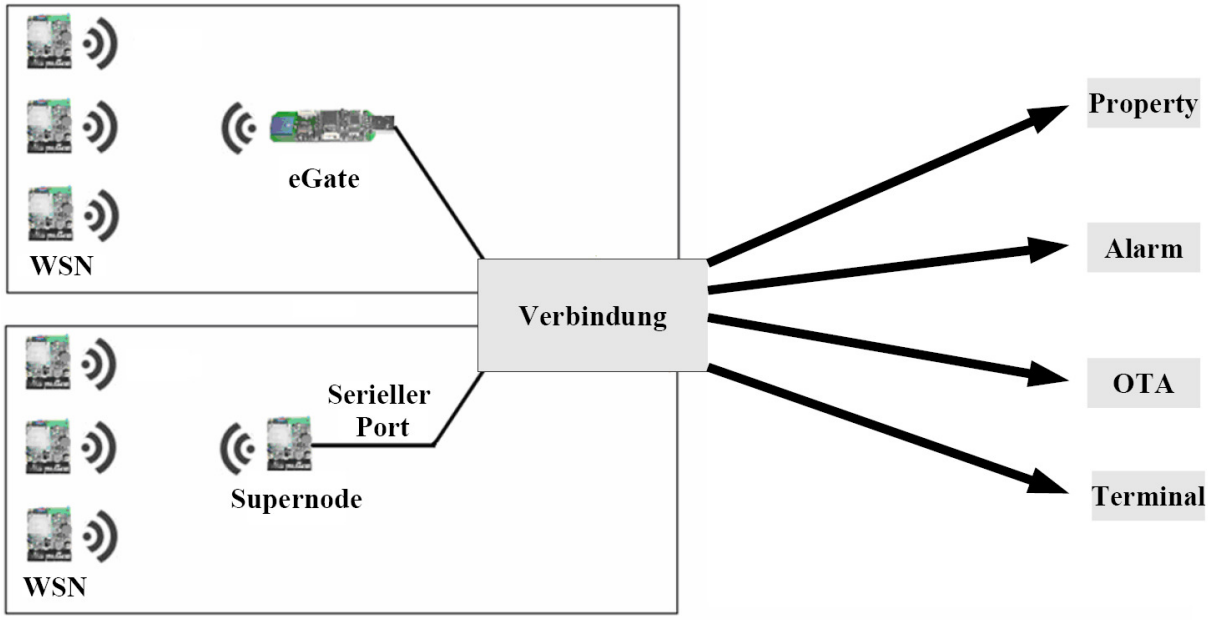


Abbildung 59 Architektur von ScatterEditor [Ding07]

Im Entwurf des Management-Plugins findet mit der Aufteilung in Reiter eine Trennung von Belangen statt, zugleich können die verschiedenen Reiter jedoch gemeinsam benutzt werden, wobei sie sich gegenseitig ergänzen und ein kohärentes Ganzes bilden. Es folgt eine Darstellung der verschiedenen Dienste bzw. Reiter des Scatter-Editors und ihrer jeweiligen Funktionalität:

1. *Connection*: Steuert den Verbindungsaufbau und die Kommunikation via eGate oder Supernode zum drahtlosen Sensornetz (WSN).
2. *Property*: Visualisierung des Status jedes sich im Betrieb befindenden Sensorknotens im WSN, zu dem im vorherigen Schritt die Verbindung aufgebaut wurde.
3. *Terminal*: Ermöglicht eine befehlsorientierte Konfiguration der Sensorknoten im WSN während ihres Betriebs.
4. *Alarm*: Ermöglicht eine benutzerdefinierte Überwachung ausgewählter Sensorknoten im Netz sowie der von ihnen erfassten Umgebung.
5. *Over The Air Flashing*: Die Aktualisierung bestimmter Softwaremodule[6] auf ausgewählten Sensorknoten im WSN.

Ein weiterer, oben nicht angeführter Dienst ist die *Server Configuration*, der nicht als Standarddienst konfiguriert ist und auf Anforderung des Benutzers aktiviert wird. Der Dienst wird somit erst zur Verfügung gestellt, wenn der Benutzer die entsprechende Einstellung im ScatterEditor betätigt, woraufhin diese laufende ScatterEditor-Instanz

[6] Diese sind in der Regel die automatisiert generierten Softwareartefakte des architekturzentrierten modellgetriebenen Prozesses in Visual ScatterUnit und ScatterFactory.

zugleich als Server anderer ScatterEditor-Klienten fungiert. Dazu bietet dieser Dienst die entsprechenden Funktionalitäten zur Vergabe von Zugriffsrechten bzw. Verwaltung der angemeldeten Klienten, die ihrerseits die Dienste von ScatterUnit mit Hilfe dieser ScatterEditor-Serverinstanz internetbasiert im entfernten Modus in Anspruch nehmen wollen. Die genaue Funktionsweise, wird weiter unten geschildert.

Wird ScatterEditor im lokalen Modus betrieben, so wird eine direkte Verbindung zum angeschlossenen eGate oder SuperNode aufgebaut. Anders ist es im Falle der internetbasierten Kommunikation, die einer Klient/Server-Architektur auf Basis der *Remote Method Invocation*[7] (RMI) von Java folgt. [WebRMI] Dabei wird das eGate bzw. der SuperNode an denjenigen Rechner angeschlossen, auf dem der lauschende RMI-Server läuft (vgl. Abbildung 60). Nun können die als Klienten laufenden ScatterEditor-Instanzen (rechts) entfernt durch die als Server laufende Scatter-Editor-Instanz (links) verwaltet werden.

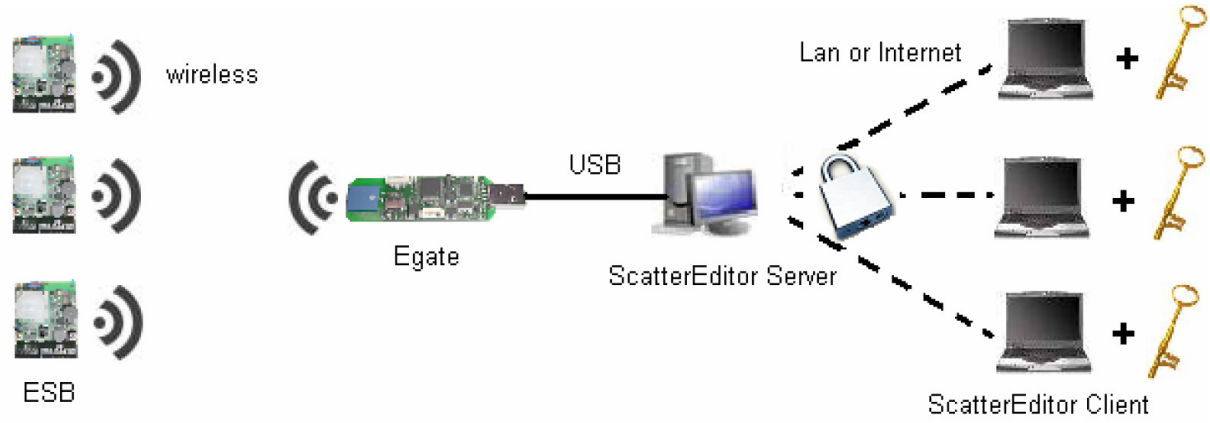


Abbildung 60 Internetbasierter Zugriff auf die Dienste von ScatterEditor [Ding07]

Jeder ScatterEditor kann entweder als Server oder als Klient fungieren. Die Konfiguration für den jeweiligen Modus wird vorgenommen, nachdem der Benutzer seine Wahl getroffen hat. Der Rechner muss im lokalen Modus laufen und das Programm RMI-registry muss gestartet werden, so dass der Computer nicht als Server benutzt werden kann. ScatterEditor läuft dann auf den korrespondierenden Klienten im entfernten Modus. Um die Verbindung herzustellen muss der Benutzer die korrekte IP und das Server-Passwort eingeben. Klienten, Zugriffsrechte und Passwörter können auf dem Server *Configuration*-Reiter verwaltet werden. Dieser Zuschnitt hat den Vorteil, dass der Benutzer beide Konfigurationsarten (Klient und Server) benutzen kann, sich aber nur mit einem Programm vertraut machen muss. Dies erleichtert den Zugang und die Akzeptanz von ScatterEditor. Die folgenden Dienste können internetbasiert im ScatterEditor in Anspruch genommen werden: *Remote_connection, Remote_terminal* und *Remote Over The Air Flashing*.

7.2. Dienste und Komponenten von ScatterEditor

7.2.1. Die Verbindungskomponente *Connection*

Abbildung 61 zeigt zwei Bildschirmabzüge des Connection-Reiters: Die Verbindungskomponente Connection fungiert als Brücke zwischen den Sensorknoten des WSNs und dem Java-basierten ScatterEditor-Rahmenwerk, durch welche die Kommunikation zwischen beiden Seiten aufrecht erhalten wird. Analog zu ScatterPlug, wird die Kommunikation zwischen beiden Seiten mittels der COM-Port-Bibliothek des javax.comm Pakets abgewickelt. Mit Hilfe der Methode eGate.connect() wird die Verbindung zum WSN aufgebaut. Hierzu müssen die entsprechenden

[7] Siehe Glossar.

Initialisierungsparameter der Methode übergeben werden, die individuell vom Benutzer mittels der Felder *Type of Communication* (1), *Communication via* (2) und *COM* (3) eingegeben werden (siehe oberer Bildschirmabzug in Abbildung 61). Anhand von (1) kann festgelegt werden, ob die Kommunikation im lokalen oder im entfernten internetbasierten Modus vonstatten gehen soll. Im Falle des internetbasierten Modus wird die Verbindung zum lauschenden Server aufgebaut, wodurch sich der Klient einen Zugang zum WSN verschafft. Andernfalls wird ein bidirektionaler textorientierter Kommunikationskanal zur Senke aufgebaut, durch den die aus- und eingehenden Nachrichten von und zum WSN ausgetauscht werden. Die Festlegung, ob es sich bei der Senke um ein EGate oder einen SuperNode handelt, wird durch (2) angegeben. Mittels (3) wird die Nummer des Ports der seriellen Schnittstelle des Rechners angegeben, an dem die Senke angeschlossen wird.

Soll der Rechner auch als Server anderer ScatterEditor Klienten gemäß der Beschreibung im letzten Abschnitt fungieren, so muss festgelegt werden, ob der Terminaldienst im lokalen oder entfernten Modus laufen soll. Falls der lokale Modus für das Terminal gewählt wird (4), werden die korrespondierenden Daten lediglich lokal auf demselben Rechner angezeigt. Ansonsten wird die entfernte Konfiguration des Terminals aktiviert und die Daten werden beim Rechner des Klienten angezeigt. Die Unterscheidung ist hier wichtig, da je nach Konfiguration des Terminals die Form festgelegt wird, in der die Daten angezeigt werden. Beim entfernten Modus werden auf Grund entfernter und somit verzögerter Kommunikation lediglich die essentiellen aus dem WSN ankommenden Daten selektiert und angezeigt und nicht jegliche aus einem Sensor auf Grund eines beliebigen Ereignisses erzeugte Nachricht.

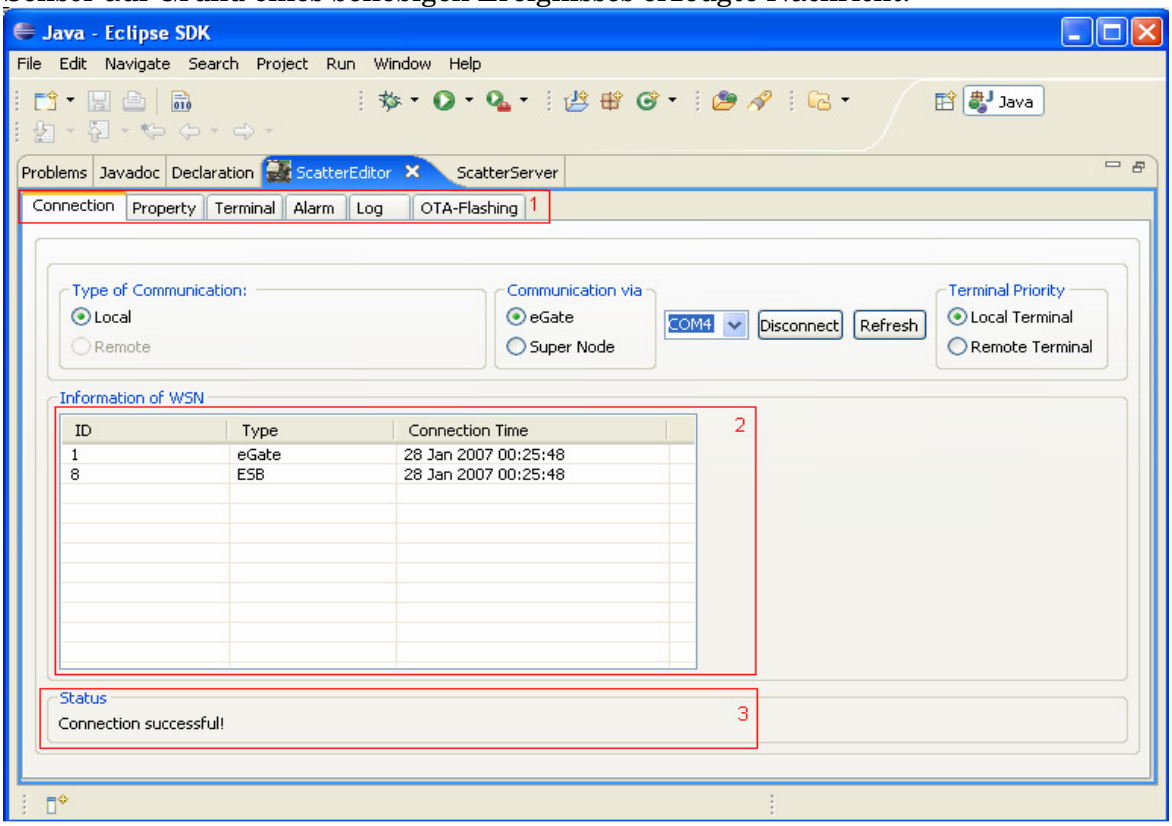


Abbildung 61 Connection-Reiter vor (oben) und nach (unten) dem Verbindungsaufbau [Ding07]

Nützliche Informationen bezüglich der Sensorknoten (wie die Sensor ID, Typ des Sensorknotens, Verbindungszeitpunkt etc.) werden in der Tabelle aufgelistet (5). Die Tabelle kann zu jedem Zeitpunkt vom Benutzer durch die *Refresh*-Funktionalität (7) aktualisiert werden, durch die alle Sensoren des WSNs neu erfasst werden. Neu dazu kommende Sensorknoten sowie nicht mehr existierende können somit flexibel ermittelt

und dem Benutzter aufgelistet werden (siehe (2) im unteren Bildschirmabzug der Abbildung 61). Der Status der Verbindung zum WSN wird mittels *Status* (6) angezeigt.

Beim Start des Plugins ist nur der *Connection*-Reiter sichtbar. Die übrigen Reiter werden erst angezeigt, wenn die Verbindung mit einem Port erfolgreich war und nachdem das WSN gescannt wurde. Dies trägt zur Übersichtlichkeit bei und erleichtert die Interaktion mit dem Benutzer. Nach einer erfolgreichen Verbindung zum WSN werden die Reiter der Dienste ScatterUnits angezeigt (1, vgl. unterer Bildschirmabzug in Abbildung 61), über die dem Benutzer der Zugriff auf die entsprechenden Funktionalitäten ermöglicht wird. Somit wird der Benutzer erst nach der erfolgreichen Verbindung zum WSN in die Lage versetzt, die Reiter zu betätigen. Dies gewährleistet, dass nur die im WSN erreichbaren bzw. aktiven Sensorknoten angesprochen werden können, wodurch unnötiger Datenaustausch vermieden wird. Im Folgenden werden die Funktionalitäten der einzelnen Reiter erläutert.

7.2.2. Visualisierung durch den Property-Reiter

Nachdem die Verbindung zum WSN erfolgreich aufgebaut wurde, wird der Benutzer mittels des *Property*-Reiters in die Lage versetzt, die Eigenschaften der im WSN aktiven Sensorknoten sowie die des eGates fortlaufend zu beobachten. Hierzu ermöglicht die *Property*-Funktionalität die grafische Darstellung des Sensorstatus sowie der durch den Sensorknoten von seiner Umgebung erfassten Daten. Dabei wird jegliches Attribut bzw. Feld der Sensormaske „on the fly“ illustriert, während durch eine transparente Benutzerinteraktion die Sensormaske beliebig konfiguriert werden kann. Abhängig von den Sensoreigenschaften können die entsprechenden Funktionalitäten vom Benutzer beliebig aktiviert bzw. deaktiviert werden. Abbildung 62 zeigt einen Bildschirmabzug des *Property*-Reiters.

Um die aktuell im WSN aktiven bzw. erreichbaren Knoten zu ermitteln, steht dem Benutzer die *Refresh* Funktionalität (2) zur Verfügung, die eine Liste der IDs aller im WSN vorhandenen aktiven Konten liefert. Aus dieser Liste (pull-down Menü) kann der Benutzer einen bestimmten Knoten wählen, mit dessen Maske die weitere Interaktion stattfindet. Wird die ID des eGates vom Benutzer gewählt, so können zusätzliche eGate-spezifische Informationen in (3) erscheinen. In der obigen Abbildung wird beispielsweise der Sensorknoten mit der ID 8 aus der Liste selektiert und dementsprechend werden die Daten seiner Maske in der grafischen Schnittstelle visuell angezeigt bzw. seine Funktionen ausgeführt. So können in (4) und (6) der Reihe nach die LEDs sowie der Pieper des Knotens betätigt werden. In (5) ist es möglich, die Konfiguration des Knoten zurück zu setzen. Die verschiedenen Firmwareattribute können in (7) und (8) vom Benutzter konfiguriert werden. Weitere technische Daten über den Sensor, wie dessen Sendestärke, externe und interne Stromversorgung, werden in (10) und (11) angezeigt. Daten über die Sensorik des Knotens, wie die Temperatur, Bewegungssensor, etc., werden in (9) angezeigt. Die visuell orientierte Gestaltung des *Property*-Reiters ist ein Beispiel dafür, wie die Mensch-Maschine-Schnittstelle (human-computer interface) für ScatterWeb realisiert wird.

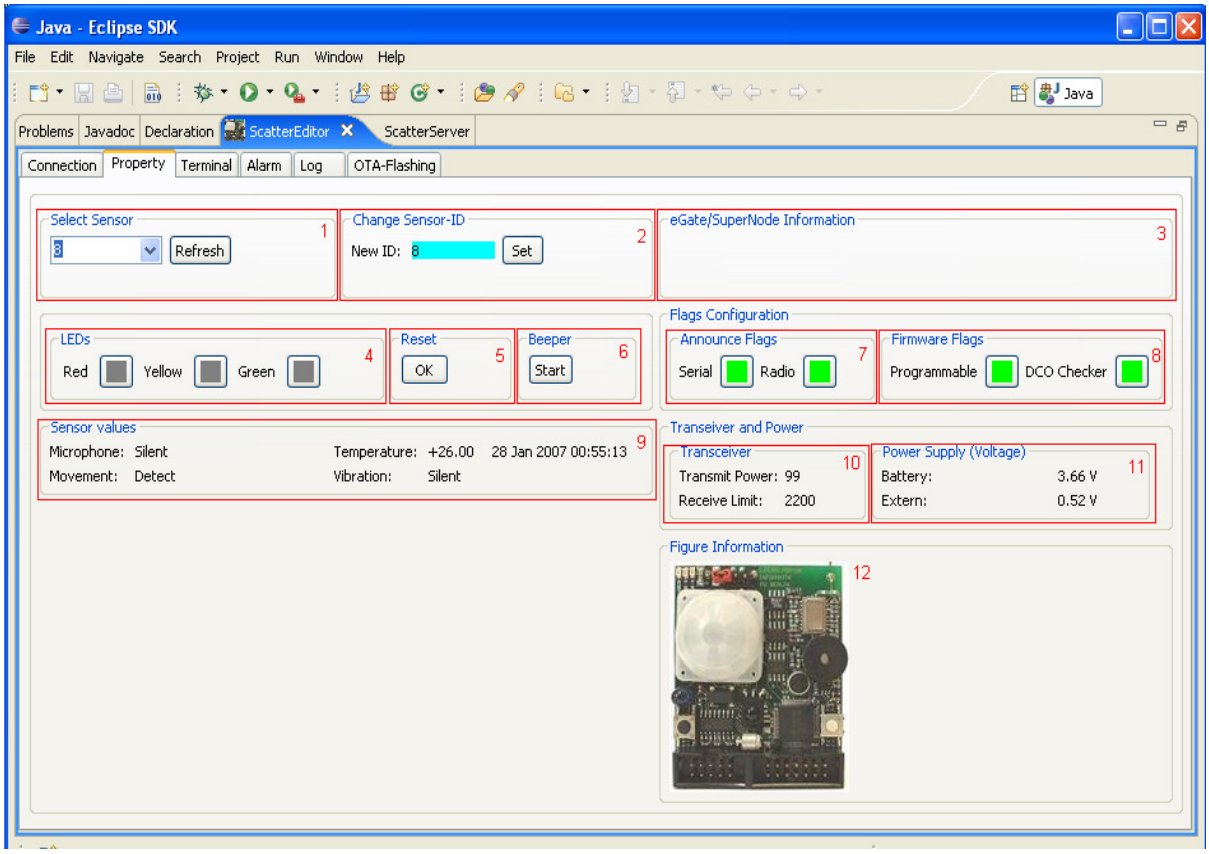


Abbildung 62 Property-Reiter [Ding07]

7.2.3. Das Terminal

Das *Terminal* bietet die Möglichkeit, die einzelnen Sensorknoten im WSN während ihres Betriebs auf einfache Weise über das eGate zu konfigurieren. Abbildung 63 zeigt einen Bildschirmabzug der Benutzerschnittstelle des Terminals.

Die Konfiguration erfolgt, indem nach einem strukturierten Muster aufgebaute Befehle an die Sensorknoten abgesetzt werden. Diese Befehle verfügen über ein spezifisches, aber einfaches Format, was ihre Formulierung erleichtert. Über diese kann der Benutzer auf den laufenden Sensorknoten interaktiv operieren, wobei für jeden Befehl die dazugehörige Meldung vom adressierten Knoten in Abhängigkeit von dessen Status versendet wird.

Das folgende Beispiel demonstriert, wie die Terminalbefehle verwendet werden:
@21 stp 99: Die ID des adressierten Knotens folgt dem @ Zeichen. Im Falle des Nichtvorhandenseins des @ Zeichens und somit auch der ID wird das eGate adressiert. Nach der Adressierung folgt dann der Befehl. Im hier behandelten Beispiel ist dies stp (Set Transmission Power), der bewirkt, dass die Sendestärke des Sensorknotens konfiguriert wird. 99 steht für den Parameter, der seinem Befehl stets folgt. Es ist außerdem möglich, einem Befehl mehrere Parameter zu übergeben.

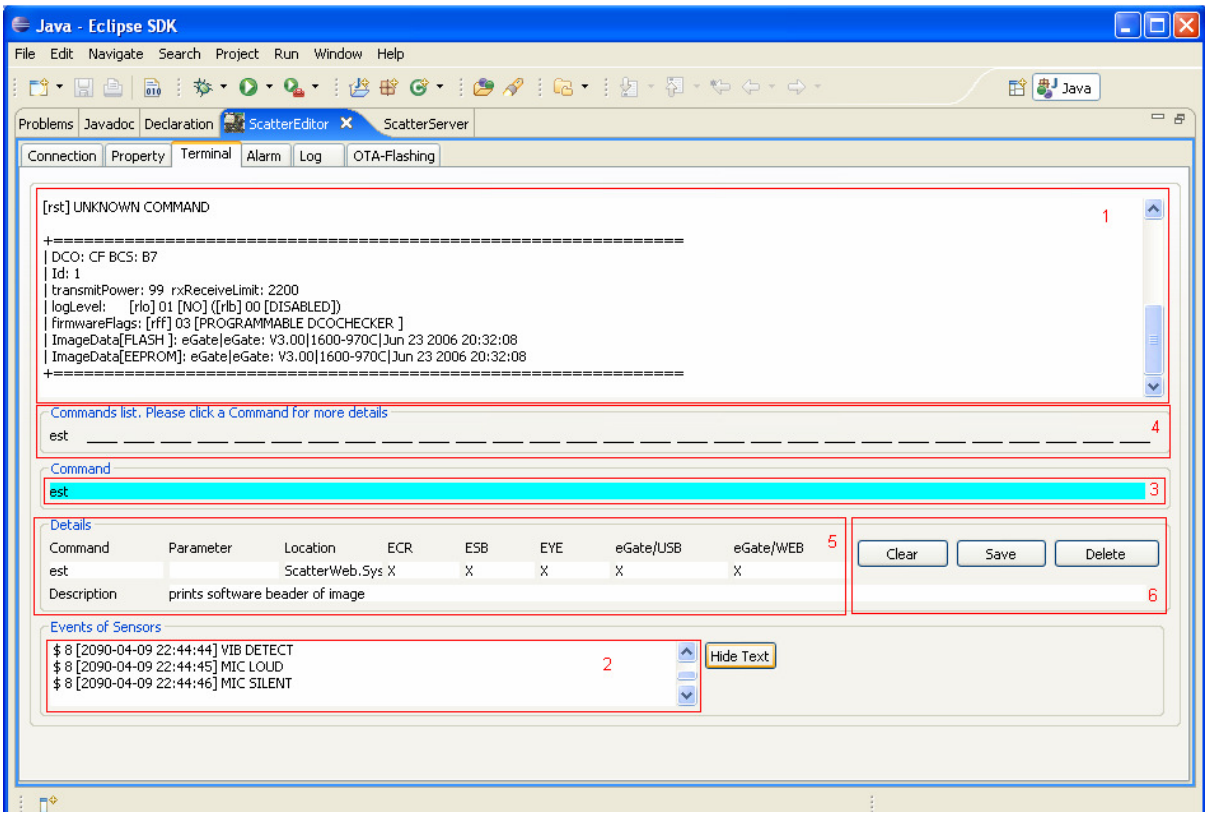


Abbildung 63 Der Terminal bietet eine befehlsgestützte Konfiguration der Sensorknoten [Ding07]

Die Befehle werden zuerst als Text (in der Form von Strings) mittels des COM-Ports[8] zum eGate geschickt. Die empfangenen Strings werden anschließend von einem spezifischen, auf dem eGate lauschenden Parsermodul interpretiert und in einem weiteren Schritt bearbeitet, wodurch für jeden interpretierten Befehl ein zugehöriges Paket erstellt wird, das abschließend zum im Befehl adressierten Knoten über den Funkkanal verschickt wird. Die Erstellung der den Befehlen entsprechenden Pakete wird ebenfalls von einem auf dem eGate befindlichen speziellen Modul realisiert. Die Funktionalität jedes Terminalbefehls wird mit einem C-Makro innerhalb der Firmware der Sensorknoten implementiert. Damit kann man individuelle Terminalbefehle flexibel definieren und durch ihre Implementierung auf C-Ebene ausführen lassen. Dies erleichtert die Durchführung von Versuchen, das Testen und die Fehlersuche bei neu implementierten Funktionen.

Das obere Ausgabefenster (1) der obigen Abbildung zeigt die Rückmeldungen der Knoten, an welche die jeweiligen Befehle abgesetzt wurden. Zusätzlich werden fortlaufend Ereignismeldungen eingeblendet (2), durch die der Benutzer ständig über wichtige Änderungen im WSN auf dem Laufenden gehalten wird. Gegebenenfalls kann der Benutzer auf das Auftreten bestimmter Ereignisse reagieren, indem er entsprechende Befehle an die betroffenen Knoten absetzt. Die Eingabe der Befehle erfolgt durch die Eingabeleiste (3). Bei der Eingabe des ersten Buchstabens während der Formulierung eines Befehls werden dem Benutzer in (4) alle mit diesem Buchstaben beginnenden Befehle aufgelistet, so dass aus dieser Liste ein beliebiger Befehl ausgewählt werden kann. Dabei werden als Zusatzinformation die Bedeutung bzw. Wirkung des selektierten Befehles sowie die ihm zu übergebenden Parameter dem Benutzer angezeigt.

Ferner ist es möglich, mittels der Tasten ↓ oder ↑ zwischen den bereits eingegebenen Befehlen zu navigieren. Zu diesem Zweck wird jeder in der Eingabeleiste abgesetzte Befehl im Hintergrund für die laufende Sitzung in einer Stapeldatenstruktur

[8] Für diesen Zweck wird wie erwähnt die Java-COM-Bibliothek des Javax-Paketes verwendet.

abgelegt. Dies wird im Schritt 1 der Abbildung 64 veranschaulicht. Im Falle, dass der Befehl durch die Betätigung der ↑ Taste wiederholt wird, wird der Befehl in der Eingabeleiste eingeblendet und zugleich aus dem Stapel A entfernt (Schritt 2) und in den Stapel B eingefügt. Wird nun die ↓ Taste gedrückt, so wird der Befehl wiederum aus dem Stapel B entfernt und in A wieder eingefügt (Schritt 4). Hierdurch wird ein schneller Zugriff auf die gespeicherten Befehle gewährleistet, durch den sie unmittelbar in der Eingabeleiste zur Verfügung stehen.

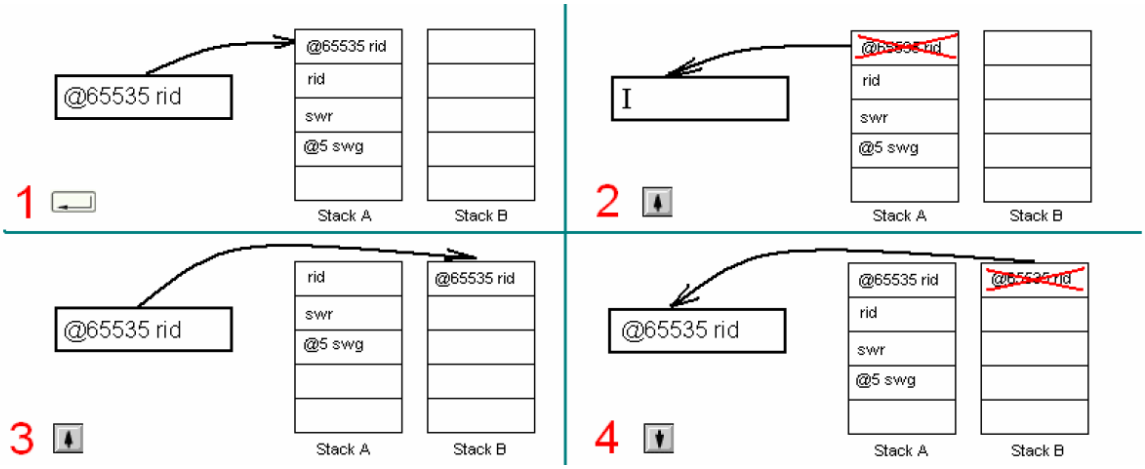


Abbildung 64 Stapeldatenstruktur zur Verwaltung der bereits eingegebenen Befehle [Ding07]

7.2.4. Aktualisierung der Softwaremodule via Funkkanal (*Over The Air Software Updates)*

Wie in 3.4 erläutert, bedingt der kognitive iterative Debuggingprozess im Rahmen des modellgetriebenen automatisierten Testprozesses eine mehrfache Verfeinerung des Testfallmodells, wobei bei jedem Verfeinerungsschritt der korrespondierende auf den Sensorknoten auszuführende Infrastrukturcode neu generiert wird. Dieser Prozeß wird durch eine flexible Ausbringung des neu generierten Codes auf die jeweiligen Sensorknoten, auf denen das (verfeinerte) Testszenario ausgeführt wird, abgerundet. Darin besteht die Funktionalität des *Over The Air Flashing*-Reiters. Ebenfalls kommt die Funktionalität des *OTA-Flashing* dem Generat von ScatterFactory zu Gute, wodurch die automatisiert generierten Codeartefakte der im Editor modellierten Anwendungen optimal an die ihnen im Modell zugewiesenen Sensoren über den Funkkanal ausgebracht werden.

Wie in 5.6 geschildert, können in einem einzigen Modell der DSL mehrere Anwendungen zusammen formal modelliert werden, wobei deren Code jeweils generiert wird. Es wäre daher sehr praktisch, wenn der Benutzer in die Lage versetzt würde, beliebig zwischen den unterschiedlichen generierten Codeartefakten zu navigieren und diese auf ausgewählte Sensorknoten auszubringen. Darin besteht das Hauptmerkmal des Entwurfs des *OTA-Flashing*-Reiters.

Die Softwareaktualisierung über Funk via eGate ist im Vergleich zum seriellen Flashing etwas komplexer. Zuerst wird eine serielle Verbindung zum eGate über den Java-COM-Port hergestellt. Für diese Aufgabe wird das Javax.com verwendet. Dadurch wird das kompilierte Softwareartefakt (Hex file) Zeile für Zeile in das EEPROM des eGates geladen. Als nächstes wird das Hex-File zu den Zielknoten per Funk gesendet und gleichzeitig auf der seriellen Verbindung zum eGate auf alle auftretenden Fehler gelauscht. Falls nötig wird dies in einer entsprechenden Dialogbox angezeigt. Eine andere Herausforderung liegt darin, dem Benutzer eine klare und einfache Interaktionsmöglichkeit mit diesem Prozess zur Verfügung zu stellen, der die

Handhabung der OTA-Flashing-Komponente in ScatterEditor ermöglicht. (Mehr zu den einzelnen Funktionalitäten der OTA-Flashing-Komponente im Anhang 11.3.3)

7.2.5. Die WSN-Überwachung (Alarm)

Drahtlose Sensornetze werden zumeist zum Zweck der Beobachtung ihrer Umgebung eingesetzt. Die Sensorknoten nehmen Messwerte aus ihrer Umgebung, bearbeiten und evaluieren diese, um Rückschlüsse auf ihre Umgebung ziehen zu können. Typisch für eine solche Messung (z.B. zum Zweck der Gebäudeautomatisierung) ist die der Temperatur. In diesem Kontext ist es sehr effizient, dem Benutzer die Möglichkeit zu geben, selbst einfache Beobachtungsbedingungen zu definieren und zu formulieren, um verschiedene Vorgänge der Umgebung nach Bedarf unter die Lupe zu nehmen.

Zur Realisierung des Alarmsystems und seinen speziellen Parsern wurde eine Alarmsprach mit einer kontextfreien Grammatik entwickelt. Die Grammatik fungiert als semantisches und syntaktisches Vehikel bzw. Gerüst, auf dessen Basis sich korrekte, klar strukturierte Alarmausdrücke formulieren lassen.

Sei G = (N, T, P, S) eine Grammatik mit N als der Menge der nicht terminalen Symbole mit S als Startsymbol und T als der Menge der terminalen Symbole, wobei P die Menge der Produktionsregeln (Herleitungsregeln) stellt, so heißt die Grammatik G kontextfrei, wenn jeder gebildete Satz aus T gemäß den Produktionsregeln aus P und beginnend mit dem Startsymbol S auf ein nicht terminales Symbol aus N zurückgeführt werden kann. Dies lässt sich mathematisch wie folgt beschreiben [Schö97, S. 13]:

$$\forall(w_1 \rightarrow w_2) \in P : (w_1 \in N) \wedge (w_1 \in (N \cup T)^*)$$

Die folgende Tabelle veranschaulicht den Aufbau der Grammatik der Alarmsprache (vgl. Tabelle 1):

a)	ALARMSYSTEM	→	alarm ID if CONSTRAINTS
b)	ID	→	int | int ID
c)	CONSTRAINTS	→	CONSTRAINT | CONSTRAINT @ CONSTRAINTS
d)	CONSTRAINT	→	CONDITION ANDOR CONSTRAINT | CONDITION TIMEPERIOD
e)	CONDITION	→	MEASUREMENT1 COMPARE float | MEASUREMENT2 detect
f)	COMPARE	→	< | ≤ | > | ≥ | = | !=
g)	ANDOR	→	and | or
h)	TIMEPERIOD	→	from TIME to TIME
i)	TIME	→	int_hour:int_min
j)	MEASUREMENT1	→	temp | batt
k)	MEASUREMENT2	→	micr | vabr | irmo

Tabelle 1 Die kontextfreie Grammatik [Ding07]

Abbildung 65 zeigt ein Beispiel des unten stehenden Alarmausdrucks und dessen Herleitung auf Basis der obigen Grammatik. Hierzu wird der zugehörige Syntaxbaum aufgestellt. Bei jedem Herleitungsschritt werden die darauf angewandten Produktionsregeln angegeben:

```
alarm 23 34 if temp ≥12 and micr detect from 12:30 to 14:50
```

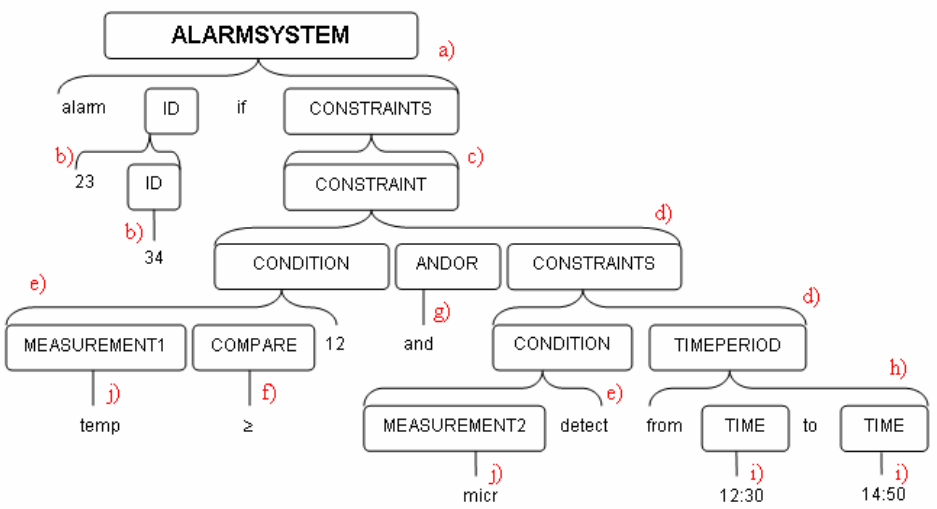


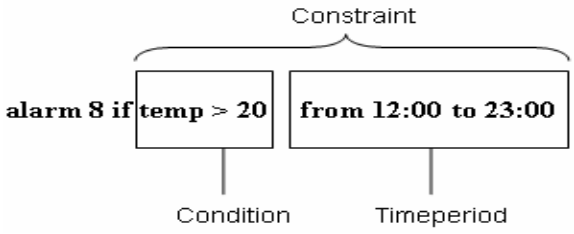
Abbildung 65 Beispiel eines Alarmausdrucks (oben) und dessen Syntaxbaum (unten) [Ding07]

Der Ausdruck gibt eine Randbedingung für den Alarm vor, so dass dem Benutzer eine Benachrichtigung geliefert wird, sobald das folgende Prädikat bei den angegebenen Sensorknoten (23 und 34) erfüllt ist: Falls eine Temperatur von einem der zwei Sensorknoten gemessen wird, die größer oder gleich 12°C ist und mindestens einer der zwei Sensoren eine Stimme während der Zeitspanne 12:30 bis 14:50 registriert.

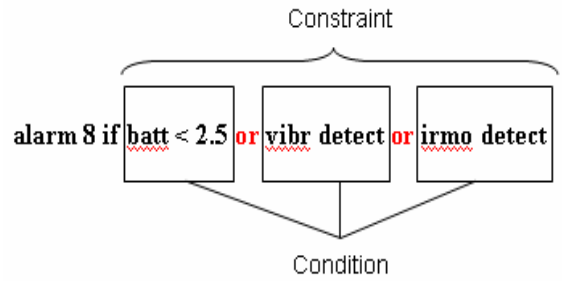
Dieser rekursive und zusammengesetzte Aufbau der Alarmsprache erleichtert dem Benutzer die Formulierung von Alarmausdrücken. Abbildung 66 zeigt drei unterschiedliche Alarmausdrücke, die demonstrieren, wie die Komposition der Alarmprädikate strukturiert ist.

1. Dieser Ausdruck steht für eine reguläre Randbedingung mit einer vorgegebenen Zeitspanne.
2. Dieser zusammengesetzte Ausdruck besteht aus der Verknüpfung mehrerer Alarmprädikate
3. Bei diesem rekursiven Ausdruck werden die Alarmrandbedingungen verkettet.

1.



2.



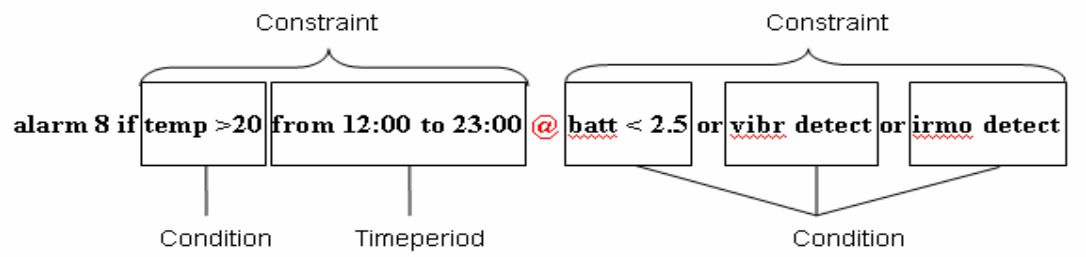
3. 

Abbildung 66 Drei rekursive Ausdrücke der Alarmsprache [Ding07]

7.3. Internetintergration in ScatterEditor

Sensornetze werden oft in Gebieten eingesetzt, zu denen Menschen nur einen begrenzten Zugang haben, wie z.B. Naturreservate oder Regionen in einem extremen Klima, u. ä. Es erscheint daher sinnvoll, ScatterEditor mit dem Internet zu verbinden, um den Zugriff auf die ScatterEditor-Dienste durch einen oder mehrere entfernte Rechner zu ermöglichen.

Zur Realisierung der Internetintegration wird – wie zu Beginn des Kapitels besprochen – eine herkömmliche IP-basierte Klient/Server-Architektur auf Basis der Java RMI-Technologie verfolgt. Zu diesem Zweck wurde ein (eGate)-Server implementiert, der auf einem entfernten Rechner läuft und für den berechtigten Klienten zugriffsbereit ist. An diesen Rechner ist auch die eGate-Hardware direkt angeschlossen. Der Server übernimmt somit die direkte Kommunikation mit dem WSN. Des Weiteren erfolgt die Verwaltung der Klientenregistrierung für das drahtlose Sensornetz zentral durch den Server.

Ausschlaggebend für einen robusten Einsatz des eGate-Servers sind die dafür notwendigen Sicherheitsmaßnahmen. Es findet eine automatisierte Vergabe von Zugriffsschlüsseln statt, um den Klienten die Berechtigung der entfernten Nutzung der Scatter-Editor-Dienste zu erteilen. Der Administrator kann nach Belieben einen neuen Schlüssel generieren. Von Seiten des Klienten wird für die Inanspruchnahme der entfernten Dienste wie folgt vorgegangen: Zuerst muss eine sichere Verbindung zum eGate-Server aufgenommen werden. Zu diesem Zweck müssen der Benutzername und das vom Server generierte Passwort eingegeben werden. Hierzu muss im *Connection*-Reiter der entfernte Modus ausgewählt werden, wodurch die Interaktion mit dem Benutzer entsprechend aktiviert wird. Dies zeigt Abbildung 67. Mehr zu Benutzerinteraktion im entfernten Modus im Anhang 11.3.5.

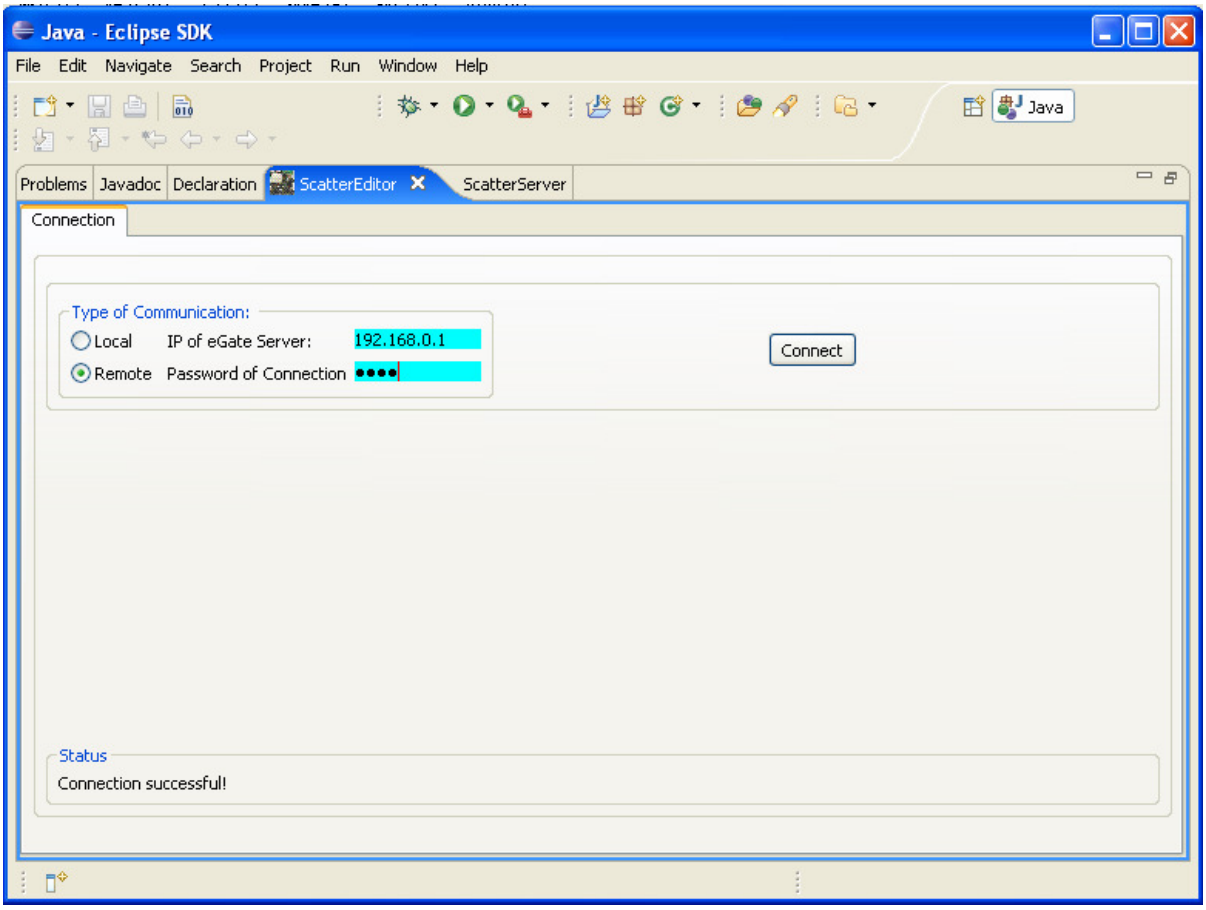


Abbildung 67 Einsatz von ScatterEditor als Klient [Ding07]

7.4. Zusammenfassung

In diesem Kapitel wurde ScatterEditor vorgestellt. Das Eclipse-basierte Rahmenwerk verfügt wie ScatterPlug über eine Plugin-orientierte Architektur und besteht aus kostenfreien Open Source-Komponenten. Neben der Integration in die Eclipse-Plattform ist es möglich, die zentralen Dienste von ScatterEditor sowohl im lokalen als auch im internetbasierten *remote*-Modus in Anspruch zu nehmen. Der webbasierte Zugriff macht es möglich, die von ScatterEditor angebotenen Dienste auf die ScatterWeb-Sensorknoten unter Bedingungen anzuwenden, die den Einsatzszenarien von WSNs sehr nahe kommen. So können die Sensorknoten an einem beliebigen Ort (dem für das Experiment gewählten Feld *Sensingfield*) in Betrieb genommen werden, während man an einem anderen Ort diese Sensorknoten illustrativ und interaktiv überwachen, programmieren und verwalten kann.

Der in Reiter (Tabfolder) aufgeteilte Aufbau bietet dem Benutzer die Möglichkeit, die jeweils benötigten Dienste so zu kombinieren, dass die sich ergänzenden Reiter in ihrer Gesamtheit als ein kohärentes Ganzes funktionieren. Die verschiedenen Reiter bieten zahlreiche Funktionalitäten zur Konfiguration, Steuerung und Überwachung der sich im Betrieb befinden Sensorknoten.

So steuert der *Connection Reiter* den Verbindungsaufbau und die Kommunikation zum drahtlosen Sensornetz. Der *Property-Reiter* ist für die Visualisierung des Status jedes sich im Betrieb befindenden Sensorknotens im WSN zuständig. Damit die Knoten auf einfache Weise während des Betriebs konfiguriert werden können, wurde der *Terminal-Reiter* entwickelt, welcher eine befehlsorientierte Konfiguration der Sensorknoten ermöglicht. Eine weitere wichtige Funktionalität des Reiter-orientierten ScatterEditors ist die Fähigkeit, die auf den Knoten laufenden Softwareartefakte per

Funk zu aktualisieren. Es wurde geschildert, wie dies mit Hilfe des Reiters *Over The Air Flashing* (OTA-Flashing) bewerkstelligt wird. Ein umfangreicher Abschnitt des Kapitels widmete sich dem Überwachungssystem, welches durch den *Alarm*-Reiter realisiert wurde. Ferner wurden die rekursive Syntax der Alarmsprache sowie deren Semantik erörtert, wobei anhand zahlreicher Alarmausdrücke illustriert wurde, wie sich Dank des strukturierten Sprachenaufbaus die Alarmausdrücke auf übersichtliche und einfache Art und Weise formulieren lassen.

8. Beurteilung und Auswertung der Werkzeugkette

Um eine umfassende und aussagekräftige Bewertung der ScatterClipse Werkzeugkette zu treffen, müssen in erster Linie die Komponenten, die das Rückgrat der gesamten Werkzeugkette ausmachen, evaluiert werden [vgl. Hent06, Kame07, Kame08]. Dabei geht es vor allem um den erzielten Effekt der angewandten Automatisierung und die Frage, wie dieser sich hinsichtlich des Einsatzes der Werkzeugkette auswirkt. Demzufolge sind an dieser Stelle zwei Aspekte von Bedeutung: die Anwendungsentwicklung und die Testautomatisierung.

Im Kontext der Anwendungsentwicklung spielt die Größe des Codes, der für die Ausführung der Anwendung notwendig ist, eine zentrale Rolle. Das Ziel ist dabei genau den Code automatisiert zu generieren, der für die Anwendung relevant ist. Zu diesem Zweck muss die Codegröße bei den durch ScatterFactory und ScatterFactory2 erstellten Anwendungen genauer untersucht werden.

Was die Testautomatisierung anbelangt, muss sich die Bewertung auf zwei Ebenen beziehen. Einerseits ist der erzielte Automatisierungsgrad bei der Erstellung der Testfälle mit Hilfe von Visual ScatterUnit von großer Wichtigkeit, anderseits ist der von ScatterUnit benötigte Verbrauch an Betriebsmitteln zur Bereitstellung der Testautomatisierungsdienste auf Sensorebene relevant. Hier muss vor allem die durch ScatterUnit verursachte Instrumentierung der Laufzeitumgebung der Sensorarchitektur genauer unter die Lupe genommen werden.

8.1. Die Codegröße der in ScatterFactory erstellten Anwendungen

Wie bereits geschildert, dient die modellgetriebene Codegenerierung bei der Anwendungsentwicklung auch der maßgeschneiderten Codeerzeugung, die nur so viel Code für die Sensorknoten erzeugen soll, wie für deren jeweilige Rolle erforderlich ist. Dadurch wird nicht nur der knappe Speicherplatz optimiert, sondern zugleich unnötige rechen- und energieintensive Softwaremodule vermieden. Der sinkende Anteil an manuellem Code reduziert zugleich die Gefahr von Flüchtigkeitsfehlern der Programmierer. Im Folgenden werden drei repräsentative, mit Hilfe von ScatterFactory erstellte Anwendungen betrachtet und dabei das Verhältnis zwischen generiertem und manuell erstelltem Code genauer untersucht.

Bei der ersten Anwendung handelt es sich um die Standard *Empty* Anwendung, die sämtliche Firmwarefunktionalitäten zur Verfügung stellt und daher die gesamten Firmwaremodule benötigt. Dies veranschaulicht das Diagramm zur modellierten Anwendung in Abbildung 68 (oben), wo der Container *Full_firmware* alle Firmwaremodule beinhaltet. Da alle Firmwaremodule in Anspruch genommen werden, kommt der Einsatz von Dummy Templates hierbei in Betracht. Die Anwendung fungiert somit als Referenz für die größte denkbare Menge des erforderlichen Codes.

Beim zweiten Szenario handelt es sich um die *YellowSwitcher*-Anwendung, die in 5.2 diskutiert wurde. Bei der dritten Anwendung geht es um eine in der WSN-Domäne geläufige Aufgabe eines Sensorknotens, bei der der Knoten im drahtlosen Sensornetz die Rolle eines Akteurs einnimmt. Die auf dem Knoten laufende Anwendung *LED* bewirkt, dass die Lampe in einem periodischen Zeitintervall leuchtet und wieder ausgeschaltet wird. Hierfür müssen Timer-Funktionalitäten der Firmware zur Zeitsteuerung zum Einsatz kommen. Dies illustriert das Diagramm der modellierten Anwendung im unteren Teil der Abbildung 68.

Die Beanspruchung des Timer-Moduls wird durch die Modellierung der entsprechenden Ports (vgl. Kapitel 5.1) erreicht. Dementsprechend werden die für die Realisierung der Timer-Funktionalität zuständigen Module in einem separaten Container repräsentiert, was zu einer höheren Strukturierung bei der Modellierung der Anwendung beiträgt.

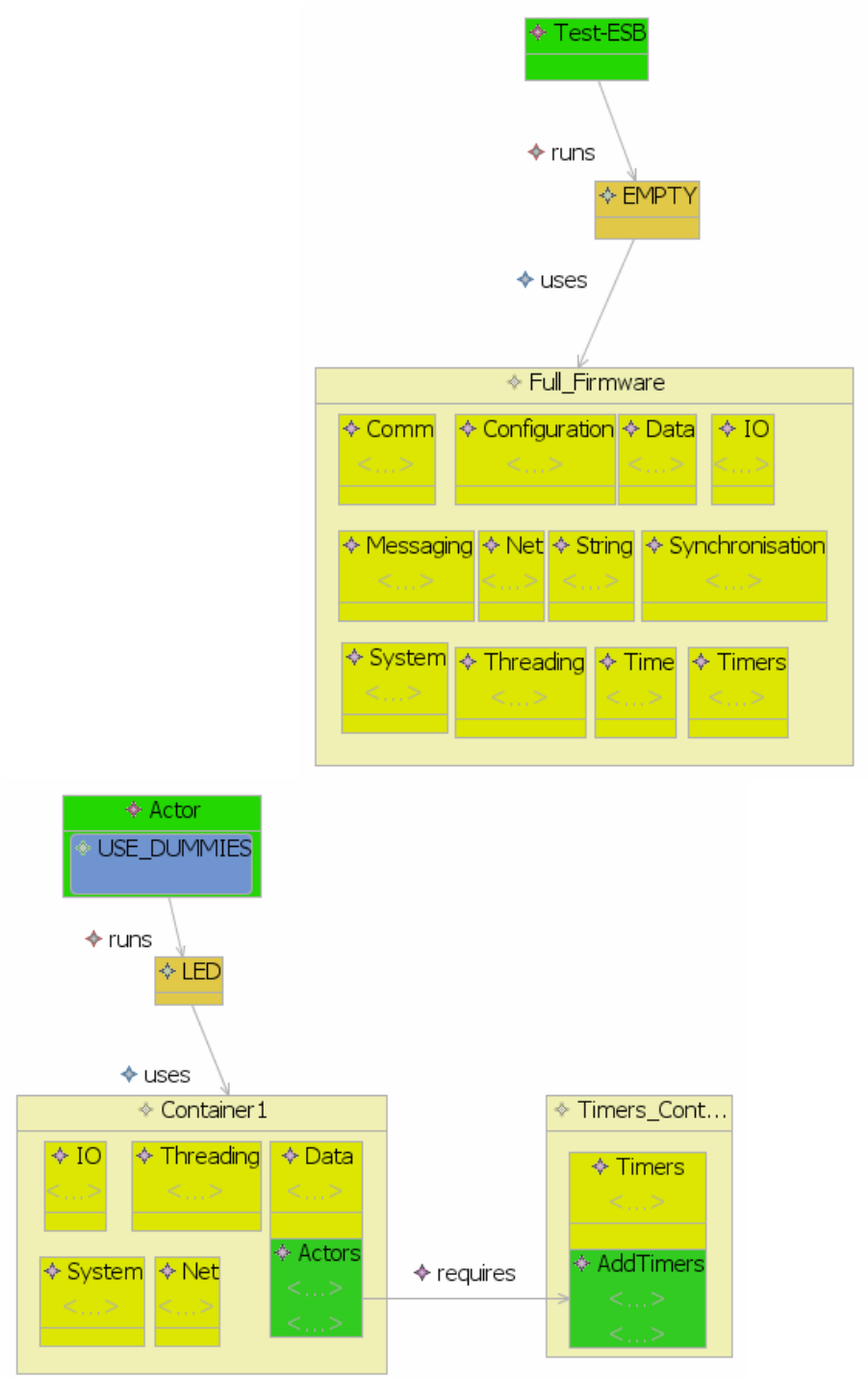


Abbildung 68 Die Modelldiagramme der Anwendungen Empty (oben) und Actor (unten) [Hent06]

Hierzu wird ein *Actor* Port für das Data-Modul in *Container1* angelegt, der seinerseits den Port *AddTimer* des anderen Containers durch die entsprechende *requires*-Assoziation in Anspruch nimmt. Die Assoziation ermöglicht, dass der Code beider Module zusammen generiert wird. Folglich wird nur dasjenige Codesegment des Data-Modules generiert, das für passives Verhalten zuständig ist. Für die restlichen Funktionalitäten des Moduls werden Dummy Templates (vgl. Kapitel 5.4) eingebunden. Gleiches gilt für das Timer-Modul, bei dem der definierte *AddTimer*-Port garantiert, dass nur die entsprechenden Templates-Ausschnitte eingebunden werden und für die sonstigen, nicht benötigten Funktionalitäten des Moduls lediglich Dummy Templates verwendet werden. Die *LED*-Anwendung steht somit für die minimale Größe an Code, die zur Realisierung einer Akteur-Rolle eines Sensorknotens benötigt wird.

Wie bereits erwähnt, ist die Codegröße maßgeblich für die Produktivität der automatisierten Anwendungsentwicklung. Tabelle 2 zeigt die benötigte Größe an generiertem Code (in Bytes) für die drei oben besprochenen Anwendungen und zwar jeweils mit und ohne die Verwendung von Filterregeln (vgl. Kapitel 5.4).

Szenario	Ohne Filter	NO_SERIAL	MIN_COOMANDS	MIN_COMMANDS
EMPTY	34050	32222	32215	31990
YellowSwitcher	24980	24230	24226	23950
LED	19846	19048	18944	18944

Tabelle 2 Größe des generierten Codes nach dem Kompilierungsprozess mit und ohne Filterregeln [Hent06]

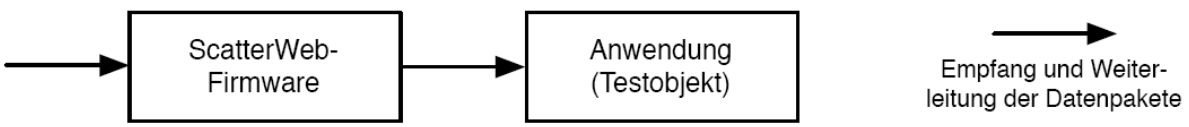
Wie der Tabelle zu entnehmen ist, kommt es bereits ohne die Aktivierung der Filterregeln zu einer erheblichen Ersparnis an generiertem Code. So ist der für die *Yelow_Switcher*-Anwendung generierte Code ohne die Benutzung von Filterregeln um ca. 27% niedriger als die Referenz, nämlich die *Empty*-Anwendung ohne Filterregeln. Mit anderen Worten wird für eine typische Anwendung eines Knotens, die daraus besteht, auf Ereignisse im WSN zu reagieren sowie empfangene Pakete zu verarbeiten, alleine durch die entsprechende strukturierte Repräsentation im formalen Modell ein beachtlicher Codeanteil eingespart. Dasselbe gilt für die zweite repräsentative Anwendung, die *LED*-Anwendung, bei der der Knoten die Rolle eines Akteurs im WSN einnimmt. Hier wird sogar eine Codeersparnis von ca. 42% erreicht. Werden nun zusätzlich die Filterregeln aktiviert, so steigert sich die Codeersparnis bei der *Yellow_Switcher*-Anwendung auf 30% und bei der *LED*-Anwendung auf ca. 45%.

Was die Beurteilung der Produktivität von ScatterFactory2 anbelangt, so lässt sich sagen, dass dank des modularen Hardwareentwurfs der MSBs und dessen kongruentem Abbild auf Metamodellebene in ScatterFactory2 nur der für die Anwendung benötigte Code mit sämtlichen dazu notwendigen Treibern generiert wird. Jegliche Bibliotheken, Treiber sowie Softwarefragmente für im Modell nicht vorkommende Sensormodule werden bei der Generierung nicht miteinbezogen. Daher wird, wie in 5.6 geschildert, bei ScatterFactory2 bei der Erzeugung der Softwarefamilien von einer automatisierten modellgetriebenen Konfiguration der Laufzeitumgebung der Anwendungen gesprochen.

Alles in allem hat sich die hier verfolgte automatisierte Codegenerierung hinsichtlich der Produktivität des zugrunde liegenden modellgetriebenen Entwicklungsprozesses der Sensorknotenanwendungen sowohl für die modulare als auch für die monolithische Infrastruktur als Optimierungstechnik zur Ressourcenausnutzung hervorragend bewährt.

8.2. Bewertung der Testautomatisierungsschicht

Wie erörtert, besteht die Testautomatisierung mit Hilfe von ScatterUnit hinsichtlich der Bearbeitung der Daten, die von der zu testenden Anwendung empfangen werden, (vgl. Abbildung 69) aus folgenden Tätigkeiten:



a) Regulärer Betrieb der Laufzeitumgebung

b) Die Verarbeitungsschritte der Zwischenschicht zur Durchführung der Testautomatisierung

Abbildung 69 Die Laufzeitumgebung der Sensorarchitektur mit (b) und ohne (a) Testautomatisierung [Kame07]

1. Herausfilterung eingehender relevanter Datenpakete für die Realisierung des Simulationsdienstes, z.B. die Änderung der Netztopologie
2. Analyse des Paketinhaltes bezüglich der Erfüllung eines im Testfallmodell definierten Ereignisses, z.B. für die Realisierung des Wartevorgangs
3. Protokollierung jedes eingehenden Datenpakets, das an die Anwendung gerichtet ist. Hierbei handelt es sich jedoch um einen nicht standardmäßig laufenden Dienst, der gesondert im grafischen Modell vom Benutzer aktiviert werden muss.

Die obigen Schritte, welche die Datenpakete zum Zweck des Anwendungstests zuerst durchlaufen müssen, werfen die Frage auf, inwieweit die Instrumentierung der ScatterUnit-Testautomatisierungsschicht das Verhalten der zu testenden Anwendung beeinflusst. Zwar handelt es sich sowohl im Kontext der ScatterWeb-Plattform als auch der ScatterClipse-Werkzeugkette um keine Echtzeitanwendungen, dennoch sollte der Betriebseffekt der Testautomatisierungsschicht minimal gehalten werden, so dass das Verhalten der zu testenden Anwendung unberührt bleibt. Hierzu spielt die durch die obigen Bearbeitungsschritte verursachte Einflussnahme eine entscheidende Rolle. Um diese zu beurteilen wurde eine Reihe relevanter Messungen der benötigten Zeit durchgeführt, die von den entsprechenden ScatterUnit-Diensten zur Realisierung der jeweiligen oben aufgelisteten Bearbeitungsschritte beansprucht wurde.[9] Die Resultate der Zeitmessungen sind in Tabelle 3 aufgelistet:

Szenario	**Gemessene Zeit**	**Zeit pro Bearbeitungsschritt**
Alle Dienste sind deaktiviert	032 ms	< 0,1 ms
Es besteht ein Wartevorgang	054 ms	0,1 ms
Die Simulation der Netzwerktopologie ist aktiviert	133 ms	0,3 ms
Wartevorgang + Simulation der Netzwerktopologie	153 ms	0,3 ms

Tabelle 3 Messung des Verarbeitungsaufwands empfangener Datenpakete in ScatterUnit [Kame07]

Die Messungen wurden in Bezug auf verschiedene Kombinationen der betroffenen ScatterUnit-Dienste vorgenommen, um unterschiedlich repräsentative Einsatzszenarien bzw. Einstellungen der Testautomatisierung bezüglich der oben besprochenen Fragestellung abzudecken.

Wie der ersten Zeile der obigen Tabelle zu entnehmen ist, dauert jeder Verarbeitungsschritt im Fall, dass sämtliche Dienste der Testautomatisierung

[9] Hierzu wurden die Verarbeitungsschritte jeweils 500-mal bezüglich der Szenarien in der Tabelle durchgeführt.

ausgeschaltet sind, weniger als eine Millisekunde. Insgesamt betrug die Zeit des Durchlaufs eines Datenpakets bis es der Anwendung verarbeitungsbereit vorliegt (siehe Abbildung 69 Teil b) weniger als 30 Millisekunden. Diese Größe fungiert als Referenz für die Steigerung des Verarbeitungsaufwands, wenn weitere Dienste zum Zweck der Testdurchführung in Anspruch genommen werden, was den Einsatzbedingungen der ScatterUnit-Testautomatisierungsschicht entspricht. So steigt die Durchlaufszeit eines Datenpakets auf 54 Millisekunden bei der Aktivierung des ereignisbasierten Wartedienstes, was eine Erhöhung von insgesamt ca. 24 Millisekunden ausmacht. Dabei bleibt die benötigte Zeit pro Verarbeitungsschritt gleich. Bei der Durchführung dieser Messung wurde auf ein Ereignis gewartet, das am Ende nicht eintrifft, und so von einem „worst case“ ausgegangen. Auf diese Weise wird garantiert, dass andere Wartevorgänge bei einer Inanspruchnahme des Wartedienstes keinen größeren Aufwand als den hier gemessenen beanspruchen. Demgemäß stellen die angegebenen Messungen eine obere Schranke des Verarbeitungsaufwands eines Datenpakets durch den Wartedienst dar.

Interessanter wird es jedoch, wenn der Wartedienst neben einem weiteren Dienst während der Testdurchführung benutzt und somit aktiviert wird. Dies ist das vierte Szenario in Tabelle 3, bei dem der Dienst zur Simulation der Netztopologie zusätzlich zum Wartedienst aktiviert wird. Die gemessene Zeit zeigt eine deutliche Steigerung auf insgesamt ca. 153 Millisekunden, wobei ca. 0,3 Millisekunden für jeden Bearbeitungsschritt benötigt werden. Ausschlaggebend ist also offensichtlich der Aufwand, den der Simulationsdienst verursacht. Daher wurde dieser alleine gemessen, was im dritten Szenario in Tabelle 3 angegeben wird. Hierfür wurde die Einbeziehung von 49 Nachbarn in der Reichweite des Knotens simuliert, was ebenfalls einen recht hohen Verarbeitungsaufwand verursacht. Die gemessene Gesamtzeit, die der Durchlauf eines Paketes benötigte, betrug 133 Millisekunden, wobei 0,3 Millisekunden für jeden Bearbeitungsschritt benötigt werden, was eine Erhöhung von 0,2 Millisekunden pro Bearbeitungsschritt ausmacht. Betrachtet man nun die minimal benötigte Gesamtzeit des Bearbeitungsaufwands von 30 Millisekunden (bei Deaktivierung sämtlicher Dienste zur Testautomatisierung) und demgegenüber die maximal gemessene Zeit des Bearbeitungsaufwands von 153 Millisekunden (bei Aktivierung des ereignisbasierten Wartedienstes und des Dienstes zur simulierten Konfiguration der Netztopologie während des gesamten Testverlaufs), so beträgt die maximale Steigerung an zeitlichem Verarbeitungsaufwand insgesamt ca. 120 Millisekunden pro relevantem Datenpaket. Dies stellt für Nichtechtzeitanwendungen eine unerhebliche Größe dar.

Neben der Verarbeitung der Pakete ist ein weiterer Einflussfaktor die Protokollierung eingehender Pakete, um das *local logging* zu bewerkstelligen. Um eine Oberschranke der möglichen Beeinflussung zu erfassen, wurde ebenfalls vom schlimmsten Fall ausgegangen und bei der Messung Protokolleinträge mit einer Datenmenge von jeweils 50 Bytes pro Log-Eintrag gewählt, was eine recht umfangreiche Größenordnung darstellt. Die Messergebnisse haben gezeigt, dass die Erstellung jedes Log-Pakets ca. 14,5 Millisekunden dauerte. Dies verursacht ebenfalls keinen bemerkenswerten Einfluss, insbesondere wenn man bedenkt, dass Log-Pakete nur in dem Fall benötigt werden, dass Log-Marken im Modell zur Verfeinerung der aufgestellten Hypothesen angegeben werden müssen. Von großer Bedeutung ist es allerdings den bei der Protokollierung, aber auch im Allgemeinen, verursachten Verbrauch im knappen Speicherplatz der Sensorknoten zu untersuchen, was im Folgenden geschieht.

Die Beanspruchung des Speichers ist gerade im Kontext der ressourcenschwachen drahtlosen Sensorknoten ein essentieller Aspekt. Für die Integration und Durchführung der Testautomatisierung innerhalb der Laufzeitumgebung (vgl. Abbildung 70) des Sensorknotens werden alle drei Speicherarten in Anspruch genommen, nämlich ROM, RAM und EEPROM. Daher befasst sich die folgende Untersuchung mit dem Verbrauch

aller dieser drei Speicherarten.[10] Die drei Speicher werden von Komponenten der erweiterten Laufzeitumgebung verwendet (die Firmware, die zu testende Anwendung, der Testfallcode und die Testautomatisierungsschicht bzw. die libScatterUnit-Bibliothek).

Tabelle 4 veranschaulicht den Speicherverbrauch der ROM und RAM-Speicher von unterschiedlich relevanten Codeartefakten, die in der erweiterten Laufzeitumgebung zur Bewerkstelligung der Testautomatisierung in Frage kommen.

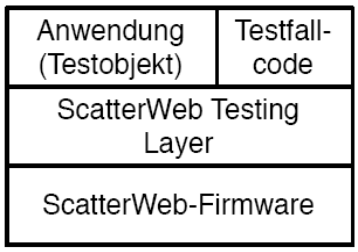


Abbildung 70 Erweiterte Laufzeitumgebung des Sensorknotens zur Testautomatisierung [Kame07]

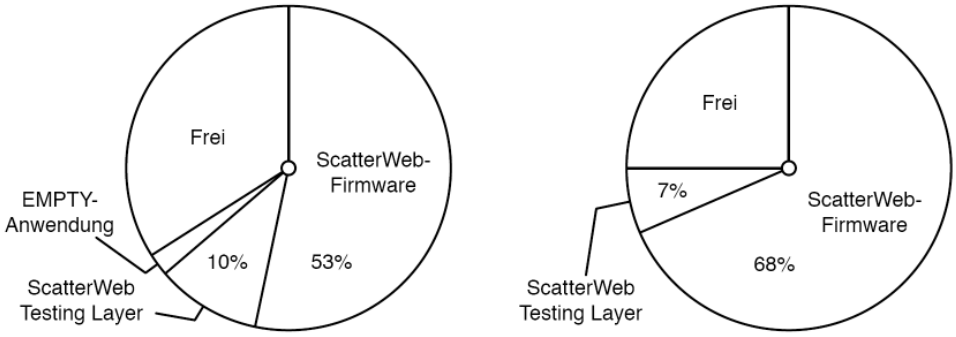
Softwaremodul	**ROM-Speicher**	**RAM-Speicher**
ScatterWeb-Firmware	032752 Bytes	001402 Bytes
ScatterWeb Testing Layer	006386 Bytes	000134 Bytes
EMPTY-Anwendung	1376 Bytes	000002 Bytes
DSR-Routing-Protokoll	3132 Bytes	000470 Bytes
Directed-Diffusion-Routing-Protokoll	21426 Bytes	000458 Bytes
Testfallcode (Beispiel)	490 Bytes	000004 Bytes
Verfügbarer Speicher	61440 Bytes	002048 Bytes

Tabelle 4 Beanspruchung der Betriebsmittel eines Sensorknotens (ESB) [Kame07]

In der Tabelle werden neben dem fixen Speicherverbrauch der Firmware und der Testautomatisierungsschicht drei unterschiedliche repräsentative Testobjekte berücksichtigt. Es handelt sich dabei um die Referenzanwendung *EMPTY*, das *Directed-Diffusion*-Ablaufprotokoll und das *DSR* Ablaufprotokoll. Während der Speicherplatzverbrauch des Testobjektes je nach Größe der zu testenden Anwendung variiert, ist er sowohl bei Firmware als auch bei der Testautomatisierungsschicht eine feste Größe, die stets in Betracht gezogen werden muss. Dies zeigt Abbildung 71 die veranschaulicht, welchen festen Speicherverbrauch bezüglich des RAM und ROM-Speichers die Firmware und die Testautomatisierungsschicht benötigen. Der übrigbleibende Speicherplatz steht dem Bedarf der zu testenden Anwendung und dem Testfallcode zur Verfügung. Wie der Grafik zu entnehmen ist, benötigt die Firmware ca. 53% und die Testautomatisierungsschicht weitere 10% des 60 KB großen ROM

[10] Die Untersuchung bezieht sich auf die ESB ScatterWeb-Sensorknoten.

Speichers. Daher bleiben ca. 37% für die zu testende Anwendung und den zugehörigen Testfallcode. Sicherlich variiert die Größe des Testfallcodes in Abhängigkeit des Testobjektes, jedoch wurde in Tabelle 4 von einer festen Größe ausgegangen (490 Bytes), die als obere Schranke für einen Testfallcode angesehen werden kann.



ROM-Speicher 60 KB **RAM-Speicher 2 KB**

Abbildung 71 Speicherplatzbedarf der Testautomatisierung [Kame07]

Denn beim Testfallcode handelt es sich überwiegend um eine Sequenz von Methodenaufrufen der Dienste der Testautomatisierungsschicht oder der Prozeduren des Testobjekts, die die Ausführung des modellierten Testszenarios veranlasst. Somit stellt der übrig bleibende Speicherplatz genügend Kapazität für die zu testende Anwendung zur Verfügung. Dies belegen die drei repräsentativ gewählten Anwendungen. Betrachtet man beispielsweise die größte der drei Anwendungen, nämlich das Directed-Diffusion-Routingprotokoll, so werden maximal 21426 Bytes beansprucht, was ca. 30% der Gesamtkapazität ausmacht.

Anders sieht es beim wesentlich kleineren RAM-Speicher aus. Hier benötigt die Firmware bereits ca. 68% der gesamten Speicherkapazität und weitere 7% kommen hinzu, die von der Testautomatisierungsschicht benötigt werden. Dadurch bleiben ca. 25% frei verfügbar, was einen knappen Wert darstellt, insbesondere, wenn man den Speicherbedarf des DSR-Routingprotokolls berücksichtigt (470 Bytes). Er ist interessanterweise etwas höher als der Speicherbedarf der wesentlich größeren Anwendung des Directed-Diffusion-Routingprotokolls, was auf die jeweilige Implementierung zurück zu führen ist. Man kann zwar durch eine optimierte Implementierung hinsichtlich der Datenorganisation auf dem *heap* und dem *stack* den RAM-Speicherbedarf reduzieren, jedoch macht dies die Realisierung der Anwendung komplexer. Alles in allem bleibt also der zwei KB kleine RAM-Speicher generell eine Herausforderung, da alleine die Firmware bereits 68% der Gesamtkapazität benötigt. Eine komfortable Lösung bieten hingegen die MSBs mit einem 5 KB großen RAM-Speicher, wodurch diese Schwierigkeit gelöst wurde. Dies bestätigen die technischen Daten der MSBs: "Since the 2 KB of RAM available on the ESB, were a very tight limitation, the MSP430F1612 with 5 KB was selected. (...) This MCU offers 60 KB of memory divided into 5 KB RAM and 55 KB Flash-ROM. (...) Instead of EEPROM an SD-/MM-card slot is included for secondary storage of up to 4 GB. It is connected to a UART and accessed using the SPI protocol." [Baar07]

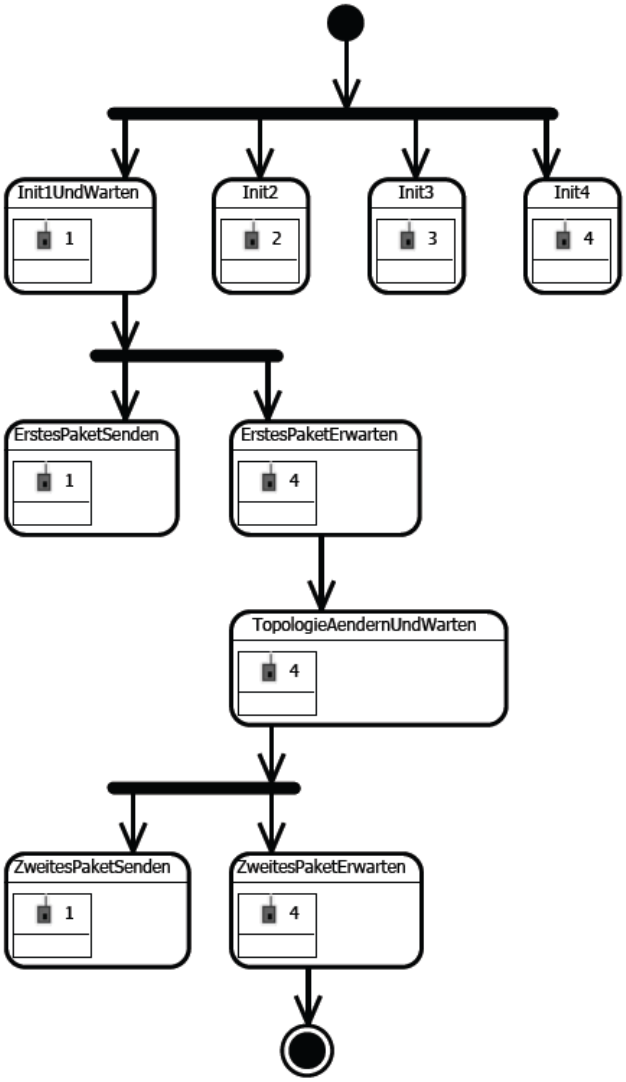
Durch die Untersuchung des Speicherplatzbedarfs der ESBs sollte illustriert werden, dass sogar bei einer knappen Speicherkapazität die Instrumentierung der Laufzeitumgebung des Sensorknotens durch die Testautomatisierung das Verhalten und die Realisierung der zu testenden Anwendung nicht beeinträchtigt. Daher bezieht sich die weitere Diskussion hinsichtlich des EEPROMS-Speicherverbrauchs ebenfalls auf die ESBs, obwohl bei den MSBs eine vier GB große SD/MM-Karte zur Verfügung steht. Das 64 KB große EEPROM wird meist zur Ablegung von Meßdaten verwendet, die von der Anwendung gesammelt werden. Daher beansprucht die Firmware einen vergleichsweise minimalen Anteil des EEPROMs. In erster Linie kommt für die Inanspruchnahme des

EEPROMs die Speicherung der erstellten Log-Einträge zur Realisierung des *local logging* zum Tragen. Für diesen Zweck wurde der Testfall für das DSR-Routingprotokoll 60 Sekunden lang ausgeführt, wobei in dieser Zeit der Protokollierungsdienst der Datenpakete voll aktiviert war. Am Ende betrug das Datenvolumen der gesamten Log-Einträge ca. vier KB, was für den 64 KB großen EEPROM Speicher keine Herausforderung darstellt. Ferner ist zu betonen, dass es sich bei den durchgeführten Testfällen keineswegs um einen Langzeitbetrieb handelt. Die Testfälle werden vielmehr gezielt hinsichtlich ausgewählter zu prüfender Fragestellungen bzw. Aspekte des Testobjektes ausgeführt. Eine Aufblähung der Log-Einträge bis zur Überschreitung der maximalen Kapazität des 64 KB großen EEPROM Speichers ist somit so gut wie ausgeschlossen. [Kame07]

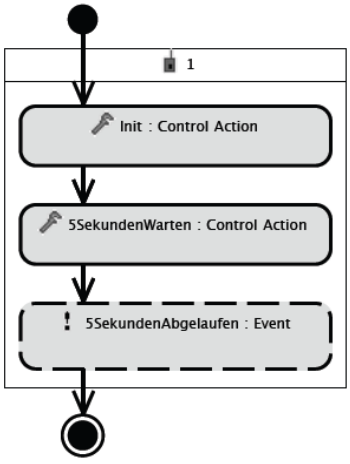
8.3. Bewertung des modellgetriebenen Anwendungstests (Fallstudie)

Zur Beurteilung der erzielten Produktivität der modellgetriebenen automatisierten Erstellung der Testfälle wurde eine Fallstudie [Kame08] durchgeführt, bei der eine Anwendung mit Hilfe von Visual ScatterUnit getestet wurde. Bei der Anwendung handelt es sich um das DSDV-Routingprotokoll (vgl. Kapitel 3.4). Wie im Folgenden gezeigt wird, war es beim modellgetriebenen Test des Routingprotokolls möglich, anhand der zahlreichen Funktionalitäten der Werkzeuge Fehler unterschiedlicher Art in der Implementierung des zu testenden Routingprotokolls ausfindig zu machen. Des Weiteren wurde die Versagensursache für jeden Fehler im Code lokalisiert und behoben, indem das Testfallmodell hinsichtlich der über die Versagensursache aufgestellten Hypothesen schrittweise verfeinert wurde bis die betroffene Stelle im Code lokalisiert werden konnte.

Die Version des in der Fallstudie getesteten Routingprotokolls wurde ursprünglich für die ESB-Knoten entwickelt. Da aber das Testwerkzeug für die zweite Knotengeneration (MSB) entwickelt wurde, war es notwendig die vorliegende Implementierung auf die MSB-Knoten zu portieren. Allerdings ging die Portierung auf Grund vorhandener Inkonsistenzen bestimmter Komponenten der beiden Plattformen nicht fehlerfrei vonstatten, was zur Folge hatte, dass die auf die MSB-Knoten angepasste Implementierung Fehler beinhaltete. Ferner wurden weitere Fehler entdeckt, die auf die initiale Implementierung zurückzuführen sind und bislang unentdeckt geblieben waren. Insofern unterliegen die Durchführung der Fallstudie und die dabei zu testende Anwendung hinsichtlich der Fehlersuche und deren Lokalisierung realen Testbedingungen, was die Aussagekraft und Authentizität der durch die Fallstudie erhaltenen Resultate deutlich erhöht. Die Abbildungen 72 und 73 veranschaulichen die während der Fallstudie entstandenen Diagramme zur Modellierung des obigen Testszenarios, das in 3.3 ausführlich erläutert wurde.

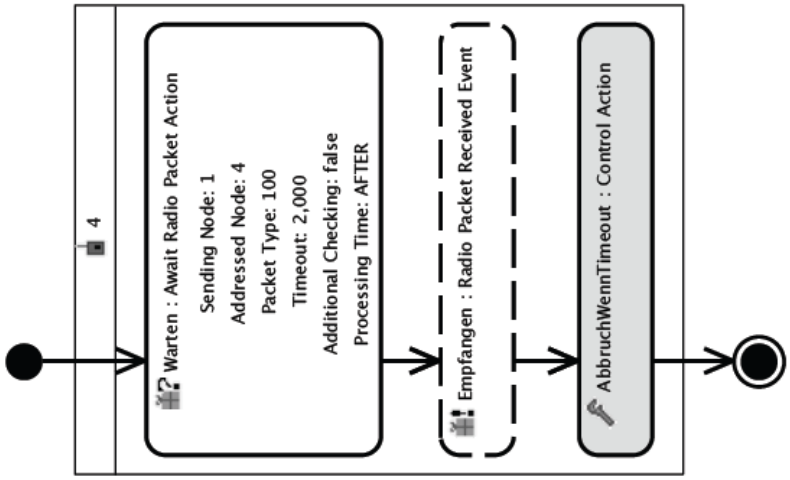


(a) Übersichtsdiagramm des Testverlaufes

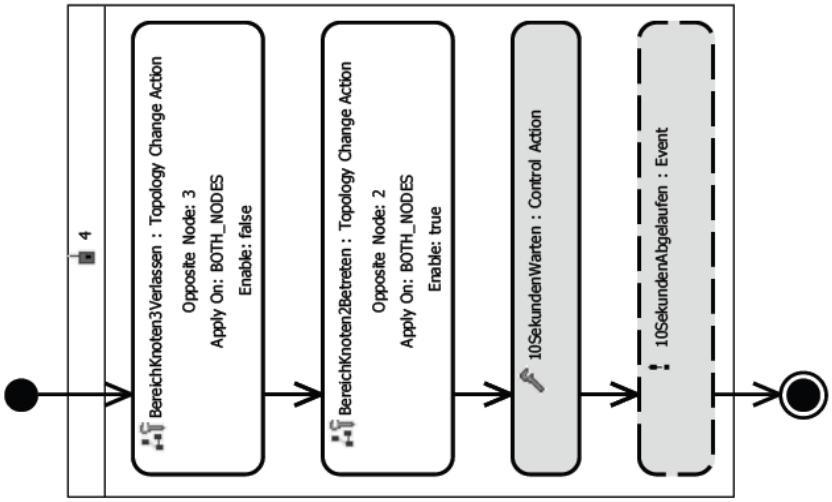


(b) Ablaufstrang-Diagramm des Ablaufstrangs Init1UndWarten

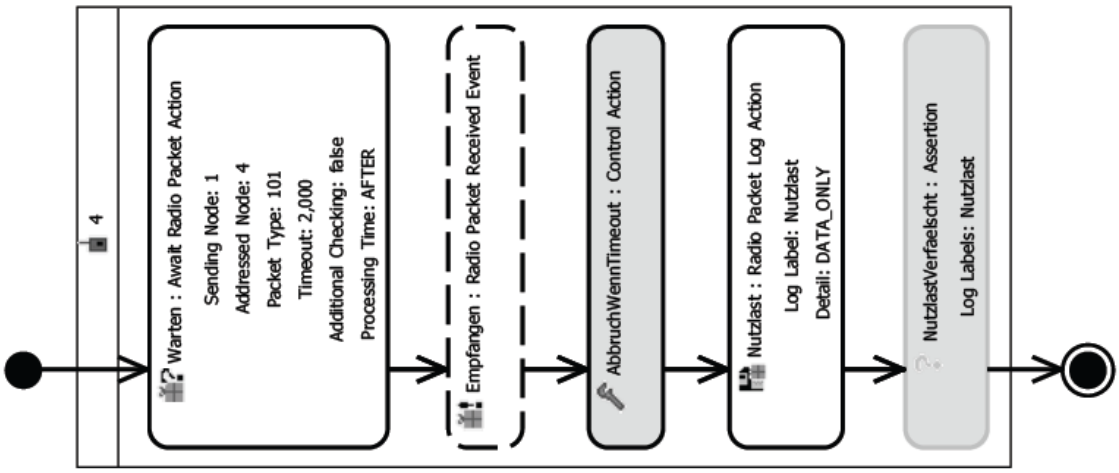
Abbildung 72 Diagramme des entwickelten Testfalls [Kame08]



(c) Ablaufstrang-Diagramm des Ablaufstrangs ErstesPaketErwarten



(d) Ablaufstrang-Diagramm des Ablaufstrangs TopologieAendernUndWarten



(e) Ablaufstrang-Diagramm des Ablaufstrangs ZweitesPaketErwarten

Abbildung 73 Diagramme des entwickelten Testfalls (Fortsetzung) [Kame08]

Die Durchführung der Fallstudie dauerte ca. zehn Stunden, wobei eine halbe Stunde für die Erstellung des Testfallmodells benötigt wurde und der Rest der Zeit für die Lokalisierung der Versagensursachen, deren Fortgang die folgenden chronologischen Schritte ausmachen. [Kame08]

Schritt 1 (benötigte Zeit: ca. 45 Minuten)

Das Versenden des Datenpakets von Knoten 1 nach 4 scheiterte. Daher hat die betroffene Methode *false* als Rückgabeparameter zurückgeliefert. Gemäß der Spezifikation der Methode wird *false* als Rückgabewert genau dann zurückgeliefert, wenn die entsprechende Routeninformation in der Routingtabelle zur Weiterleitung des Pakets bis zum Empfänger nicht korrekt vorliegen. Zu diesem Zweck wurde die erste Verfeinerung des Testfallmodells vorgenommen, indem eine Log-Aktion (vgl. Abbildung 74) definiert wurde, um die Routingtabelle zum Zeitpunkt der Versendung des Pakets zu protokollieren. Anhand der protokollierten Informationen der Routingtabelle stellte sich heraus, dass die betroffenen Informationen in der Tabelle vorhanden waren. Dies impliziert, dass das Versagen auf deren Interpretation bzw. Bearbeitung zurück zu führen ist.

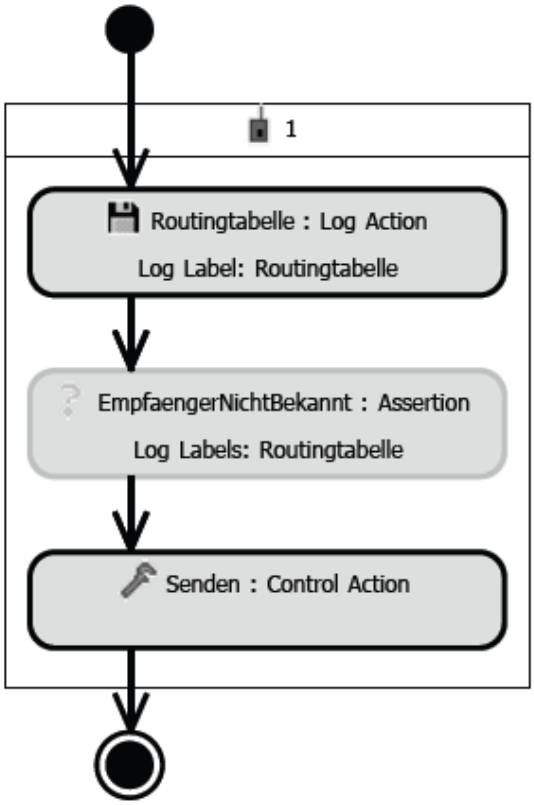


Abbildung 74 Protokollierung und Analyse der Routingtabelle zum Zeitpunkt der Versendung des Paketes [Kame08]

Schritt 2 (benötigte Zeit: ca. 10 Minuten)

Weiterhin war es nicht möglich, das Datenpaket erfolgreich zu versenden, obwohl die benötigten Einträge in der Routingtabelle zum Zeitpunkt der Versendung des Pakets korrekt vorlagen. Um sicher zu stellen, dass die Einträge auch fehlerfrei von der betroffenen Methode bearbeitet werden, wurde der 17-zeilige Code der Methode genauer untersucht. Dabei wurde die Versagensursache für den gescheiterten Versand des Pakets bestimmt. Der Typ der Empfängeradresse, die der Methode als formaler Parameter übergeben wird, stimmte nicht mit dem der Adressen der Routingtabelle

überein. Die Typunverträglichkeit ist auf einen Fehler bei der Portierung des zu testenden DSDV-Routingprotokoll von der ESB- zur MSB-Plattform zurückzuführen. Bei den ESBs werden zwei Bytes Adressen verwendet, für die Adressierung bei den MSBs hingegen nur ein Byte.

Schritt 3 (benötigte Zeit: ca. 75 Minuten)

Nun gelang das Versenden des Pakets vom Sender aus, es wurde jedoch festgestellt, dass der Zielknoten das Paket trotzdem nicht empfing. Aus diesem Grund wurde der erste Schritt erneut in Betracht gezogen, wobei der Fokus auf den *Next Hop*-Informationen der Routingtabelle lag. Diese Überlegung wurde dadurch motiviert, dass der Sendevorgang des Datenpakets erfolgreich ausgeführt werden konnte, das Datenpaket beim Empfänger jedoch nicht ankam, was darauf hinweist, dass der Defekt bei der Weiterleitung des Pakets entlang der Route liegt. Daher wurde eine weitere Verfeinerung des Testfallmodells vorgenommen, indem eine Prüfroutine zur Überprüfung der *Next Hop*-Informationen in der Tabelle (vgl. Abbildung 75) hinzugefügt wurde. Nach dieser Modellergänzung wurde der Testfall erneut durchgeführt und es wurde anhand der neu definierten Prüfroutine tatsächlich ein fehlerhafter Eintrag bezüglich des *Next Hops* festgestellt. Die aufgestellte Hypothese hatte sich somit bewahrheitet.

Schritt 4 (benötigte Zeit: ca. 90 Minuten)

Die Beobachtung im vorherigen Schritt führte zur Erkenntnis, dass das Versagen entweder an dem Austausch der Informationen unter den Routingtabellen der beteiligten Knoten oder an der Verarbeitung, wie dem Auslesen bzw. Eintragen der Tabelleninformationen liegen muss. Diese Erkenntnis bestimmte den Verlauf der weiteren Schritte zur Durchführung des Testfalls um die Fehler zu lokalisieren.

So wurde zunächst das Testfallmodell um eine weitere Prüfroutine namens *WarteAufRoutingInfo* ergänzt (vgl. Abbildung 76), die ein Anhalten im sendenden Knoten (Knoten 1) bewirkt und zwar genau während der Initialisierungsphase des Routingprotokolls, da in dieser Phase der Austausch der Routingtabelleninformationen zum ersten Mal erfolgt. Dieser Wartevorgang soll sicherstellen, dass der sendende Knoten unmittelbar vor dem Versenden des Pakets über die korrekten Routingeinträge verfügt. Wie das Diagramm in Abbildung 75 veranschaulicht, bezieht sich die Prüfroutine auf die *Next Hop*-Informationen bezüglich des Empfängers (Knoten 4). Daher wird auf die ankommenden Routinginformationen von Knoten 3 (dem erwarteten *Next Hop*) nach der Topologieänderung gewartet.

Bei einer mehrfachen Durchführung des ergänzten Testfalls wurde jedes Mal festgestellt, dass die Ausführung des Testfalls nicht terminiert. Das Voranschreiten des Testverlaufs wurde stets dort blockiert, wo die Wartefunktionalität eintritt. Schließlich wurde nach mehrfacher Begutachtung der betroffenen Stelle, die zur Blockierung der Ausführung führte, festgestellt, dass Informationen vom Knoten 3 nicht an den Knoten 1 adressiert wurden. Dies verursachte das endlose Warten auf die nicht ankommenden Pakete. Dies führte unmittelbar zur Ursache, nämlich, dass die vorliegende Implementierung des Routingprotokolls zum Datenaustausch die *Broadcast*-Addressierung verwendet. Demzufolge wurde der Wert des Parameters *Adressed Node* des Wartevorgangs von 1 (Empfänger-Adresse) auf die Broadcast-Adresse 255 gesetzt.

Nun terminierte die Ausführung des Testfalls nach erneuter Durchführung, und mittels der Prüfroutine wurde festgestellt, dass der sendende Knoten 1 die korrekten *Next Hop*-Informationen erhalten hatte. Ferner wurde beobachtet, dass das versendete Paket beim Empfänger (Knoten 4) erfolgreich angekommen war.

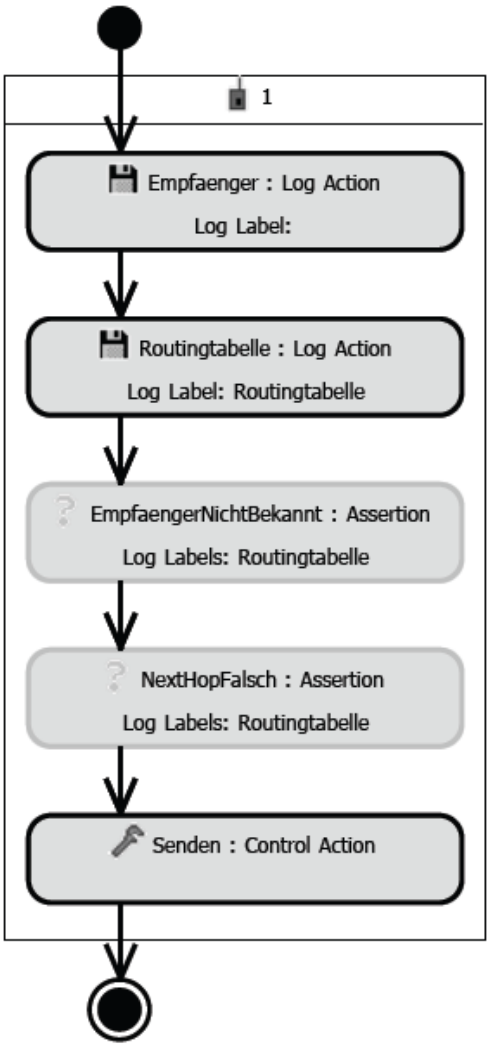


Abbildung 75 Ergänzung des Detaildiagramm „ErstesPaketSenden“ um eine Prüfroutine zur Überprüfung der Next-Hop-Information für Schritt 3 [Kame08]

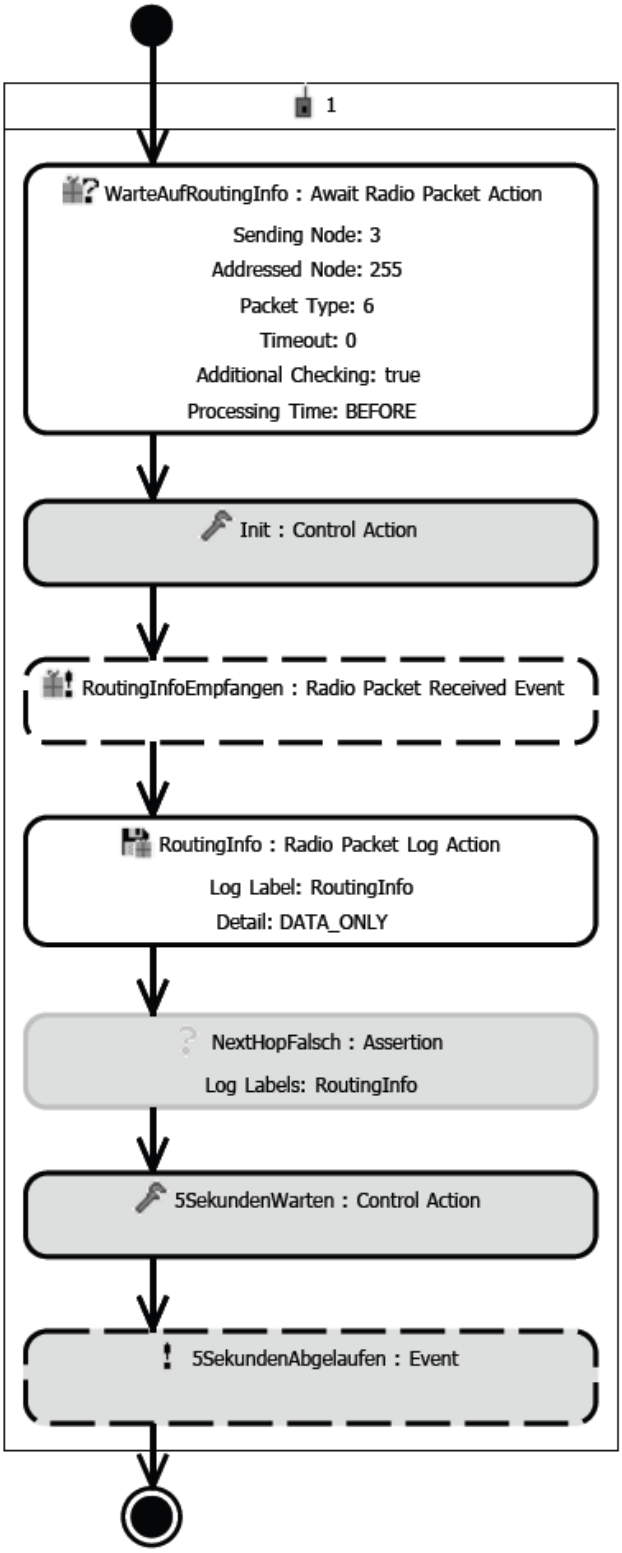


Abbildung 76 Ergänzung zum Detaildiagramm *Init1UndWarten* (Warten auf den Empfang der Routing-Information über den Empfänger des Datenpakets für Schritt 4) [Kame08]

Schritt 5 (benötigte Zeit: ca. 60 Minuten)

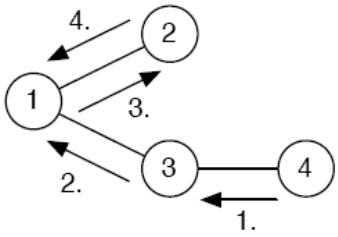
Der Testfall wurde aus Gründen der Robustheit mehrfach durchgeführt und leider wurde paradoxerweise festgestellt, dass das Versenden stets klappte, das versendete

Datenpaket den Empfänger jedoch nicht immer erreichte, da die Routingtabelle nur zeitweise die korrekten Einträge beinhaltete. Um nun die Gewissheit zu erreichen, dass tatsächlich falsche Routinginformationen vorlagen, wurde eine weitere Wartefunktion neben der ersten definiert. Die zweite Wartefunktion bewirkt das Anhalten des Testverlaufs auf dem betroffenen Knoten 1 zu einem späteren Zeitpunkt im Testverlauf. Auch in diesem Fall wurden nach wie vor manchmal falsche Routinginformationen festgestellt. (Der Schritt war nicht unbedingt notwendig, jedoch konnte damit mehr Gewissheit erzielt erden, bevor im Testverlauf weiter verfahren wurde.)

Schritt 6 (benötigte Zeit: ca. 90 Minuten)

Der aus den bisherigen Schritten resultierende Stand der Kenntnis über die Routinginformationen ist, dass obwohl die korrekten Routinginformationen ausgetauscht wurden, trotzdem manchmal nicht übereinstimmende Einträge in der Routingtabelle vorliegen. Die Ursache des Versagens muss daher in der Verbindung zwischen dem Informationsaustausch und der entsprechenden Aktualisierung der Routingtabellen gesucht werden. Ausschlaggebend für die Aufstellung dieser Hypothese ist der geschilderte Umstand, dass die Routinginformationen zeitweise korrekt in der Tabelle vorliegen, sich dann ihr Wert aber unerwartet zu einem falschen ändert, was zum Versagen führt. Dies ist ein deutlicher Hinweis darauf, dass der korrekte Wert spontan falsch überschrieben wird. Daher müsste die Routinginformation zusätzlich zu einem weiteren Zeitpunkt des voranschreitenden Routingvorgangs auf ihre Korrektheit geprüft werden. Gemäß dem Routingprotokoll verhält sich jeder Knoten während des Austauschs der Routinginformationen wie folgt: Beim Empfang einer Routinginformation trägt der Knoten sich selbst als Next Hop ein und inkrementiert im Gegenzug die Distanz. Anschließend leitet er die aktualisierten Einträge per Broadcast weiter. Demzufolge weist das Routingprotokoll den folgenden Austausch von relevanten Routinginformationen auf (vgl. Abbildung 77).

Zuerst verschickt der Zielknoten die eigenen Routinginformationen per Broadcast an alle in seiner Reichweite befindlichen Knoten (1). Der benachbarte Knoten 3 deklariert sich seinerseits als Next Hop zum Zielknoten und inkrementiert die Distanz. Diese aktualisierten Informationen leitet er anschließend ebenfalls per Broadcast an die restlichen Knoten weiter (2). Knoten 1 (Sender des Datenpakets) empfängt nun die aktualisierten Routinginformationen und ändert die eigenen Routinginformationen entsprechend ab. Sein bisheriger Kenntnisstand wird gemäß dem Protokoll per Broadcast versendet und so von Knoten 2 empfangen, der dann seine Anträge entsprechend anpasst (3). Nachdem Knoten 3 seine Routinginformationen aktualisiert hat, verschickt er sie per Broadcast weiter, was zur Folge hat, dass Knoten 1 (der Sender) die Routinginformationen bezüglich des Empfängers (Knoten 4) erneut empfängt (4).



Schritt	Empfänger	Next Hop	Distanz
1	4	4	0
2	4	3	1
3	4	1	2
4	4	2	3

Abbildung 77 Austausch der Routinginformationen über den Zielknoten (Knoten 4) [Kame08]

Schritt 7 (benötigte Zeit: ca. 75 Minuten)

Die kritische Frage ist nun, ob Knoten 1 die bereits in Schritt zwei aktualisierten korrekten Routinginformationen, mit denen von Schritt vier fälschlich überschreibt. Denn eine erneute Überschreibung würde zu einem nicht korrekten Eintrag bezüglich des Next Hops führen, bei dem Knoten 2 als Next Hop erfasst wird. Um diese Befürchtung in eine aussagekräftige Hypothese zu fassen, wurde das zuletzt erweiterte Detaildiagramm *Init1UndWarte* erneut um eine Log-Funktionalität ergänzt (*RoutingTabelleneintragDavor* und *RoutingTabelleneintragDanach*), wodurch der Eintrag vor und nach dem Empfang der betroffenen Routinginformationen protokolliert wird (vgl. Abbildung 78). Die Beobachtung nach mehrfacher Durchführung des erweiterten Testfalls ergab, dass die Routinginformationen nicht überschrieben wurden. Somit wurde die aufgestellte Hypothese nicht erfüllt. Der Fokus der Fehlersuche muss daher auf andere Aspekte gerichtet werden, um neue Hypothesen aufstellen zu können. Daher wurde zunächst der Protokollierungsdienst ankommender Pakete von ScatterUnit aktiviert, um sicherzustellen, dass die versendeten Routinginformationen korrekt beim Empfänger ankommen.

Nach der Durchführung des Testfalls mit aktiviertem Dienst wurden keine falschen ankommenden Routinginformationen festgestellt. Nun stellte sich die Frage, warum die betroffenen Routinginformationen beim Sender des Datenpakets verfälscht werden, obwohl sie bei ihm korrekt ankommen. Diese Fragestellung legte nahe, als nächstes zu untersuchen, welche Eigenschaften bzw. Muster die falschen Einträge aufwiesen, anstatt zuerst zu überprüfen, wie es zur unerwünschten Datenmanipulation kam. Daher wurde der Testfall mehrfach durchgeführt und jeweils die Einträge der Routingtabelle des Senders (Knoten 1) genauer unter die Lupe genommen. Dies war möglich, da bereits in Schritt drei das Detaildiagramm um die entsprechende Log-Funktionalität erweitert wurde, mit der die gesamte Tabelle protokolliert werden kann. Nach jedem Testdurchlauf wurde festgestellt, dass der Eintrag zum *Next Hop* immer dann, wenn Knoten 3 und 4 die Empfänger sind, mit dem Wert der eigenen Sensor-ID beschrieben wurde. Dies veranschaulicht Tabelle 5:

Empfänger	Next Hop	Distanz
1	1	0
2	2	1
3	1	1
4	1	2

Tabelle 5 Tabelleninformationen nach erneuter Testdurchführung [Kame08]

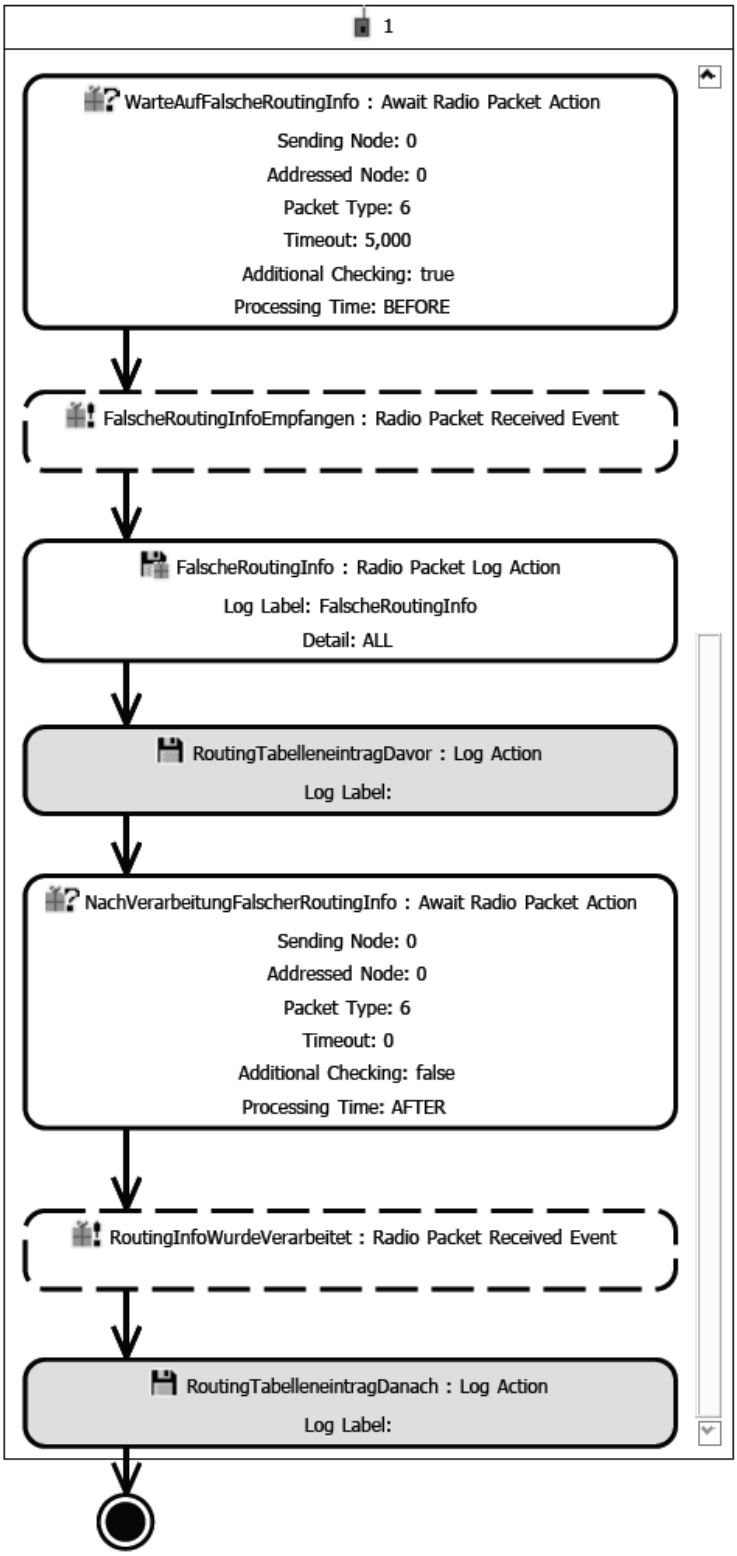


Abbildung 78 Das erweiterte Detaildiagramm *Init1UndWarten* für die Protokollierung des Eintrags der Routingtabelle vor und nach dem Empfang der betroffenen Routinginformation [Kame08]

Schritt 8 (benötigte Zeit: ca. 45 Minuten)

Nun wurde der Typus der verfälschten Informationen bestimmt, was einen wichtigen Schritt für die Lokalisierung des Fehlers bedeutet. Dabei lag der Fokus auf dem relevanten Codefragment, das für das Beschreiben der Next Hop-Information in der

Tabelle zuständig ist. Nach intensiver Analyse der Codezeilen wurden die Fehler gefunden. Die zwei Fehler, auf die das Versagen zurückzuführen war, bestanden darin, dass die von einem Knoten empfangenen Routinginformationen in Bezug auf Knoten 3 und 4 zwar wie vorgesehen vom Empfängerknoten zuerst bearbeitet wurden, um diese dann per Broadcast weiterzuleiten. Jedoch wurden der Broadcast-Methode jeweils die Einträge der Tabelle selbst übergeben und nicht die bereits bearbeiteten Kopien der Datenstruktur der empfangenen Routinginformation. Das Auffinden von Fehlern dieser Art in einem fremden Code mit Hilfe von Visual ScatterUnit ist eine weitere Demonstration des effektiven Einsatzes der hier vorgestellten visuellen kognitiven Testautomatisierung.

8.3.1. Analyse und Evaluierung der Fallstudie

Zunächst muss eine adäquate Skala bestimmt werden, anhand der die aus der Anwendung des modellgetriebenen Paradigmas gewonnene Produktivität hinsichtlich des Testprozesses bemessen werden kann. Zu diesem Zweck wurde die in der Fallstudie getestete Anwendung separat manuell mit Hilfe der Dienste von ScatterUnit getestet. Demzufolge wurden keine Testfallmodelle eingesetzt, aus denen Codeartefakte jeglicher Art generiert werden. Sämtliche Dienste von ScatterUnit mussten manuell aufgerufen werden. Des Weiteren musste der gesamte Code zur Koordinierung des dezentralen Testverlaufs von Hand implementiert werden. Ferner wurden sämtliche Log-Einträge des Testverlaufs ebenfalls manuell vom Benutzer analysiert. Es findet keinerlei Visualisierung des Testverlaufs bzw. des Verhaltens des Testobjekts statt. Die unautomatisierte herkömmliche Testdurchführung fungiert somit als Referenz, gegen welche die aus der Automatisierung resultierende Leistungssteigerung gemessen wird.

Schon bei der Konzeption des Testszenarios wurde klar, dass ohne den Einsatz formaler Modelle große Hindernisse bei der Umsetzung des Testentwurfs in einen Testfall entstehen, der den Anforderungen des Testszenarios Rechnung trägt. Die Testbarkeit der Anwendung wird generell erschwert, da man ohne die durch die Modelle erzielte Abstraktion zwangsläufig mit den technischen Belangen der Plattform konfrontiert wird. Dadurch entfernt sich der Fokus von der eigentlichen Problemstellung, nämlich dem optimalen Entwurf für einen Testverlauf, aus dem sich im Laufe der Testdurchführung präzise Hypothesen formulieren lassen, mit denen die Versagensursache lokalisiert werden kann.

Zwar bietet die manuelle Testdurchführung nur eine eingeschränkte Palette an Funktionalitäten hinsichtlich der Fehlersuche, jedoch wurden schließlich sämtliche Fehler in der Implementierung des Routingprotokolls aufgefunden. Sicherlich spielt hier eine wesentliche Rolle, dass die Fehler durch die zuvor durchgeführte Fallstudie bereits bekannt waren. Dennoch wurden aussagekräftige Erkenntnisse gewonnen, gerade wenn man den Aufwand der manuellen Testdurchführung mit der modellgetriebenen Testdurchführung vergleicht. Dies zeigt Abbildung 79, in der angegeben wird, wie viele Codezeilen notwendig waren, um den Testfall des gleichen Testobjekts einmal manuell und einmal automatisiert durchzuführen.

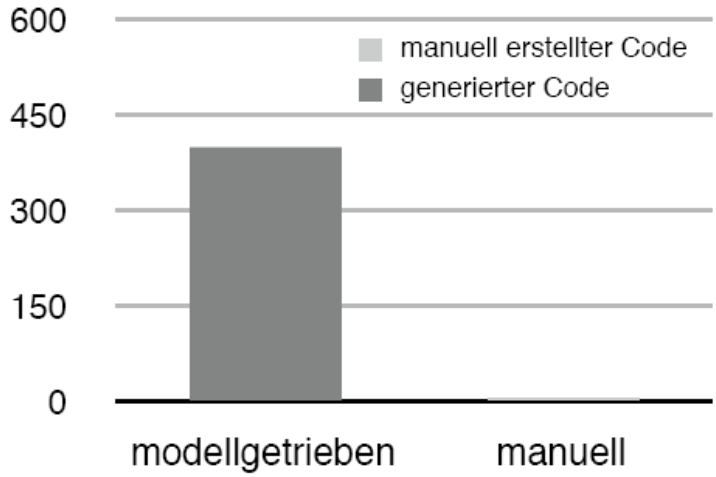


Abbildung 79 Benötigte Codegröße (in Codezeilen) für die Testfallentwicklung [Kame08]

Betrachtet man den benötigten Code beider Varianten, wird schnell ersichtlich, dass bei der modellgetriebenen automatisierten Variante insgesamt wesentlich mehr Code notwendig ist. Dies ist jedoch kein Indiz dafür, dass der Test dadurch aufwendiger wird. Der größere Codeanteil ist darauf zurückzuführen, dass bei der modellgetriebenen Testdurchführung etliche Funktionalitäten zur Verfügung stehen, die bei der manuellen Variante nicht vorhanden sind. So geht die gesamte Bearbeitung und Analyse der gesammelten Log-Einträge voll automatisiert vonstatten. Des Weiteren wird die Visualisierung des Verhaltens des Testobjekts auf Basis der zuvor analysierten Log-Einträge ebenfalls modellgetrieben und somit automatisiert realisiert. Es ist weiterhin hervorzuheben, dass diese automatisiert generierten zusätzlichen Codeartefakte ausschließlich auf dem Werkzeug und nicht auf den Sensorknoten ausgeführt werden. Daher tragen solche Funktionalitäten in jedem Fall zu einer gesteigerten Leistungsfähigkeit bei, ungeachtet dessen, welcher Codeanteil für ihre Realisierung nötig ist.

Nun stellt sich die Frage, inwieweit sich die Fehlersuche im modellgetriebenen Visual ScatterUnit automatisieren lässt, d.h. ob sich die in ScatterUnit angebotenen Dienste (vgl. 4.4) zur automatisierten Testdurchführung auf Sensorebene selbst automatisieren lassen. Solche Codeartefakte werden ausschließlich auf den Sensorknoten ausgeführt. Wie die Grafik in Abbildung 80 veranschaulicht, konnten nahezu alle Codeartefakte zur Fehlersuche automatisiert generiert werden. Bei dem kleinen nicht automatisierbaren Codeanteil (ca. 5%) handelt es sich um anwendungsspezifischen Code. Alle restlichen Codeartefakte dienen der Automatisierung der Fehlersuche. Dabei handelt es sich um domänenspezifischen Infrastrukturcode, der an den Codegenerator delegiert und somit dem Benutzer abgenommen wird. Auf diese Weise konnte der architekturzentrierte Ansatz optimal eingesetzt werden. Der vorliegende Problemraum stellt eine Domäne mit sich oft wiederholendem Infrastrukturcode dar, die sich zweckmäßig architekturzentriert automatisieren lässt. Dies bestätigt auch der benötigte Code der manuellen Testdurchführung in seiner Gesamtheit. Denn hier handelt es sich vor allem um den für die Koordinierung des dezentralen Testverlaufs zuständigen (Infrastruktur-)Code, welcher vollkommen aus seinem zugehörigen visuellen Modell im modellgetriebenen Visual ScatterUnit generiert werden kann.

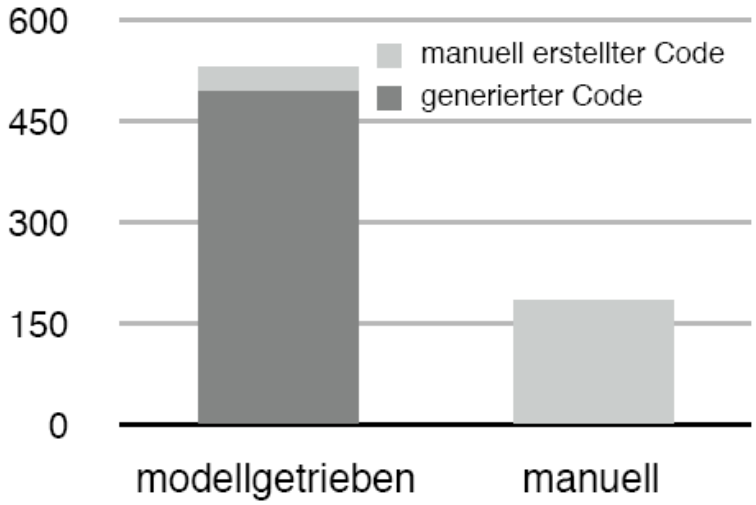


Abbildung 80 Benötigte Codezeilen für die automatisierte Fehlersuche [Kame08]

Es bleibt zu untersuchen, wie sich die Automatisierbarkeit des benötigten Codes während des Voranschreitens der Testdurchführung nach der Fehlersuche verhält. Ausschlaggebend ist dabei, wie sich die Fehlerlokalisierung bei der Fehlersuche auf das Verhalten der Codegenerierbarkeit im Laufe des Testverlaufs auswirkt. Insofern ist es sinnvoll, den Automatisierungsgrad hinsichtlich der schrittweisen Verfeinerung des Testfallmodells zu betrachten. Demzufolge ist das Verhältnis zwischen manuellem und generiertem Codeanteil in Bezug auf die einzelnen Schritte der Fallstudie näher zu untersuchen. Dies zeigt Abbildung 81, welche veranschaulicht, inwiefern die iterative Verfeinerung des Testfallmodells zur Hypothesenbildung einen Anstieg an Codezeilen mit sich bringt. Dabei wird das Verhältnis zwischen manuellem und generiertem Code angegeben. Ferner zeigt die Abbildung, wie sich dieser Codeanstieg auf die jeweiligen in der Fallstudie durchgeführten Schritte verteilt.

Wie man sieht, wird der Anteil an nicht automatisierbarem Code klein gehalten. Der gesamte Codezuwachs ist zwar durchaus beachtlich, jedoch wird der zusätzlich entstandene Code überwiegend automatisiert generiert. Der manuelle Codeanteil ist vor allem auf die Realisierung der Programmlogik der im jeweils verfeinerten Modell definierten Log-Aktionen und Prüfroutinen zurückzuführen. Bemerkenswert ist, dass bei den Schritten zwei, fünf und sechs (vgl. Abbildung 81) keinerlei manueller Code benötigt wurde, da bei allen drei Schritten das Modell derart verfeinert wurde, dass lediglich Standarddienste von ScatterUnit in Anspruch genommen werden mussten, welche vollständig automatisiert generiert werden können. Hingegen wurde insbesondere in Schritt drei verhältnismäßig viel manueller Code benötigt, da in diesem Schritt die Verfeinerung des Modells daraus bestand, das bisherige Diagramm um neue, recht große Log-Aktionen und Prüfroutinen zu ergänzen, welche anwendungsspezifisch sind. Dennoch bleibt der Anteil an nicht automatisierbarem, jedoch visuell modellierbarem Code nach mehrfacher Modellverfeinerung im modellgetriebenen Visual ScatterUnit insgesamt wesentlich geringer als der von Hand zu implementierende Code, wenn der Test manuell und ohne jegliche visuelle Modellierung durchgeführt wird.

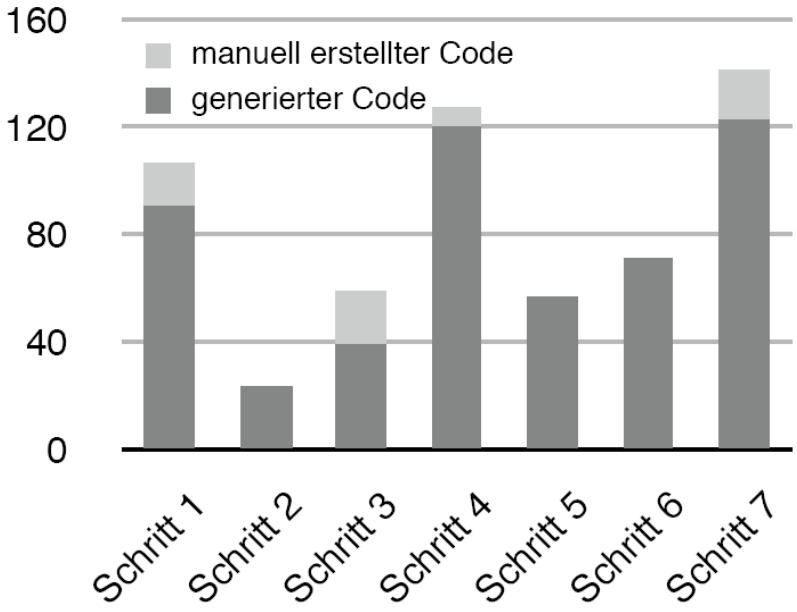


Abbildung 81 Anstieg des benötigten Codes[11] (hinzugefügte Codezeilen) zur Fehlersuche vor und nach der Fehlerlokalisierung und der mehrfachen Verfeinerung des Testfallmodells [Kame08]

8.3.2. Schlussbemerkung zur Fallstudie

Wie die Fallstudie zeigt, nimmt die Größe des Testfalls während der Fehlerlokalisierung signifikant zu. Zeit und Aufwand, die dafür benötigt werden, können durch den hohen Anteil an generiertem Code eingespart werden. Der große Anteil des generierten Codes, der vor allem für die Ausführung des Testfalls verwendet wird, bedeutet einen weiteren Vorteil. Er stellt sicher, dass die Aktionen in der korrekten Reihenfolge ausgeführt werden. Im Testfallmodell kann diese Sequenz auf klare und verständliche Art und Weise modelliert werden, wodurch Fehler in dieser Sequenz sehr viel weniger wahrscheinlich sind, als in einer manuell modellierten Sequenz.

Folgende Aufgaben wurden, abgesehen von der Erweiterung des Testfalls, durchgeführt, um die Fehler zu lokalisieren: Die Ausführung des Testfalls (die wenig Zeit in Anspruch nahm), Analyse der Testergebnisse und daraus hervorgehend die Aufstellung von Hypothesen über die Fehlerursache und deren Überprüfung. Darüber hinaus wurde das Ablaufprotokoll durchgesehen, um seine Funktionsweise zu verstehen. Besonders zeitintensiv war es, das Protokoll durchzusehen und Hypothesen aufzustellen. Der Grund hierfür liegt darin, dass die Überprüfung der Hypothesen und die Erweiterung des Testfalls so schnell vonstatten ging. Die Ursache dafür liegt im Einsatz der Codegenerierung: Für die Erweiterung des Testfalls wird lediglich eine einfache Erweiterung des Testfallmodells und die folgende Hinzufügung eines kleinen Anteils von manuellem Code benötigt. Da der Testfall-Code die Testergebnisse produziert und im Testfallmodell visualisiert wird, blieb der Zeitaufwand für die Überprüfung der Hypothesen gering, da die Testergebnisse ohne Schwierigkeiten gelesen werden konnten.

Die modellgetriebene Testumgebung macht daher eine zeitsparende Testfallentwicklung möglich, indem sie einen großen Teil des Codes generiert. Die Testfallentwicklung weist darüber hinaus eine Robustheit gegenüber Fehlern auf, weil der Code erst nach der Validierung von einem Testfallmodell generiert wird, so dass Fehler zu einem früheren Zeitpunkt entdeckt werden. Wie die Fallstudie zeigt, werden

[11] Hier handelt es sich wie in Abbildung 80 um die Codeartefakte zur automatisierten Fehlersuche, welche auf den Sensorknoten ausgeführt werden.

diese Vorteile bereits während der Entwicklung eines einfachen Testfalls deutlich. Die Nützlichkeit erweist sich jedoch besonders während der Fehlersuche, da in dieser Phase der Testfall ständig neu ausgeführt und erweitert wird.

Bei der bisherigen Evaluierung handelt es sich eher um eine funktionale Beurteilung der ScatterClipse-Werkzeuge. Eine qualitative Beurteilung ihrer Eigenschaften erfolgt im Zusammenhang der Darstellung verwandter Arbeiten im nächsten Kapitel, um den spezifischen Beitrag von ScatterClipse im Forschungsspektrum darzulegen.

9. Verwandte Arbeiten

Auf der ICSNC-Konferenz [WebICSNC] im Oktober 2008, auf welcher der größte Teil der ScatterClipse-Werkzeugkette weitestgehend vorgestellt wurden, wurde interessanterweise von Teng et. al eine Studie [Teng08] über die in der aktuellen Forschung vorhandenen repräsentativen Werkzeuge und Umgebungen zur Entwicklung von Softwareanwendungen für drahtlose Sensorknoten präsentiert und die jeweiligen Ansätze verglichen. Hierzu wurden zunächst die an solche Werkzeuge zu stellenden Anforderungen formuliert, um auf dieser Basis einen Vergleich zwischen den Werkzeugen bzw. Ansätzen vornehmen zu können.

Zunächst stellt die Unterstützung einer ausdruckstarken *Modellierungstechnik* eine essentielle Anforderung an solche Werkzeuge dar, wodurch eine Abstraktion von technischen Belangen bewirkt wird. Ferner wird erwartet, dass die Werkzeuge *Simulationsfunktionalitäten*, mit deren Hilfe gewisse Abläufe simulativ vonstatten gehen können, zur Verfügung stellen, um den Entwicklungsprozess leistungsfähiger zu gestalten. *Überwachungsfunktionalitäten*, durch die das Verhalten der Knoten und deren Umgebung näher beobachtet werden kann, werden ebenso als unerläßlich eingestuft. Weiterhin wird ein adäquater *Kompilierungsprozess*, der zusätzlich durch *Debugging-Funktionalitäten* unterstützt wird, als unverzichtbar für die Robustheit und die korrekte Ausführung der zu entwickelnden Anwendungen angesehen. Eine Reihe weiterer Komponenten wird genannt, die die Architektur der Werkzeuge mitliefern muss, um den Entwicklungsprozess abzurunden: Die Werkzeuge sollen den entsprechenden *Editor* für die visuelle Modellierung bzw. textuelle Anwendungscodierung mitliefern, die *Ausbringung* der Codeartefakte auf die Sensorknoten ermöglichen, einen *Projekt-Manager* zur Verwaltung der Dateien der erstellten Anwendungen sowie einen *Burner* zur Verfügung stellen, mit dem aus den verschiedenen kompilierten Quelldateien ein gemeinsames ausführbares Codegefüge (Code-Image) automatisiert erstellt werden kann. Dies ist die Voraussetzung dafür, die Codeartefakte auf die jeweiligen Sensorknoten ausbringen zu können. Im Folgenden werden die einzelnen Werkzeuge, die in der Studie berücksichtigt wurden, beurteilt.

Burr et al. von der ETH Zürich stellten im Juni 2006 YETI vor. [Burr06] YETI ist analog zu ScatterClipse eine auf Basis von Eclipse erstellte, Plugin-orientierte Entwicklungsumgebung, die speziell für das TinyOS [WebTinyOS] WSN-Betriebssystem entwickelt wurde. Nebst Standardmerkmalen einer Entwicklungsumgebung (gerade aufgrund der Eclipse-Integration) wie CVS-Unterstützung, Projekt-Manager, nesC[12]-Editor sowie Kompilierung und Ausbringung der entwickelten Codeartefakte, ist die größte Leistung der Entwicklungsumgebung die grafische Darstellung der aktuell zu entwickelnden Anwendungen auf Basis des komponentenbasierten TinyOS: „This tool produces a graphical representation of the currently developed application and can be used to plot the relation between its modules. Exploiting the hierarchical structure of TinyOS modules the user can decide on the graph's level of abstraction by expanding or collapsing some of the elements. If required it is possible to expand the graph to show all modules forming the current application including the ones of the operating system.“ [Burr06]

Zwar bietet die grafische Darstellung eine visuelle Darstellung der zu entwickelnden Anwendung, jedoch findet hier keine Abstraktion in der Hinsicht statt, dass die visuelle Darstellung auch als formale Modellierung fungiert, aus der die zugehörigen Codeartefakte automatisiert generiert werden könnten. Dies wäre eine sehr

[12] Dies ist die Programmiersprache des TinyOS Betriebssystems. Es handelt sich um eine leichte Erweiterung der herkömmlichen Programmiersprache C zur Einbettung komponentenbasierter Eigenschaften in die Programmstruktur. Für nesC wird daher ein spezieller Compiler benötigt. Mehr dazu in [Gay03].

zweckmäßige Erweiterung der Entwicklungsumgebung. Der Test der zu entwickelnden Anwendung wird ebenfalls nicht unterstützt. Ferner fehlen Verwaltungsfunktionalitäten wie die Konfiguration und Visualisierung des Sensorstatus sowie die funkgestützte Ausbringung der Softwaremodule. Eine Internetintegration in die Eclipse-Plattform zur entfernten Betätigung der Umgebungsmerkmale wird ebenfalls nicht unterstützt.

Eine weitere Eclipse-basierte Entwicklungsumgebung ist TinyDT [WebSall], die an der Vanderbilt Universität entwickelt wurde. TinyDT wurde ebenfalls für die Unterstützung des TinyOS-Betriebssystems entwickelt. „The most important feature that distinguishes TinyDT from the existing TinyOS IDE attempts is that TinyDT's internal parser builds an in-memory representation of the actual nesC application, which includes component hierarchy, wirings, interfaces and the JavaDoc style nesC documentation." [WebSall] Insofern ähnelt sie sehr dem zuvor besprochenen YETI-Rahmenwerk. Allerdings bietet TinyDT im Gegensatz zu YETI keine grafische Darstellung der TinyOS-Komponenten der jeweils zu entwickelnden Anwendung.

GRATIS, *Graphical Development Environment for TinyOS*, [WebGRATIS] ist ein ebenfalls an der Vanderbild Universität entwickeltes Rahmenwerk für die modellgetriebene Entwicklung von Anwendungen drahtloser Sensorknoten. Da es in der Studie berücksichtigt wurde, soll es an dieser Stelle behandelt werden und nicht in dem Abschnitt, der sich mit modellgetriebenen Werkzeugen für Anwendungsentwicklung beschäftigt. GRATIS befindet sich mittlerweile in der zweiten Version. Die Modellierungsmethode folgt der klassischen modellbasierten Entwicklung, wobei aus den formalen Modellen der zugehörige Code automatisiert für TinyOS und die darauf gestützten Anwendungen generiert werden kann. Die Entwickler bezeichnen ihr Werkzeug zwar als modellbasiert, allerdings ist es auch als modellgetrieben einzustufen, da der Code aus den formalen Modellen automatisiert generiert wird. „OS and application component interfaces along with their interdependencies are captured in a graphical environment and the glue code that ties together the application and OS components are automatically generated." [WebGRATIS]. Ferner geschieht die Modellierung gemäß dem modellgetriebenen Paradigma auf Basis einer komponentenbasierten Domänensprache, für die ein Metamodell und ein spezieller Editor entwickelt wurden.

Beide sind mit Hilfe von *Generic Modeling Environment* (GME) [WebGME] entwickelt worden, mit deren Hilfe Metamodelle zur Entwicklung domänenspezifischer Sprachen und deren zugehörige Editoren realisiert werden können. Aus eigener, in der Lehre gewonnener empirischer Erfahrung mit GME stellt das robuste Rahmenwerk starke Merkmale für den modellgetriebenen Prozess zur Verfügung, insbesondere auf der Ebene der Metamodellierung und der grafischen Editoren der Domänensprachen. Ferner findet eine Validierung der formalen Modelle statt, bevor die Codegenerierung angestoßen wird. Dies ist dem Constraint Manager zu verdanken, mit dessen Hilfe OCL-basierte Randbedingungen vom Benutzer definiert werden können. Es fehlt jedoch eine Integration mächtiger Codegeneratoren, was dazu führt, dass der Codegenerierungsprozess eingeschränkt bleibt. Ferner ist eine Integration in die Standardentwicklungsumgebung nicht ausreichend vorhanden.[13] Aus diesem Grund musste für GRATIS ein gesondertes Eclipse-Plugin entwickelt werden, um von den Merkmalen der Eclipse-Plattform Gebrauch machen zu können. Zwar schneidet GRATIS bei der Studie (wie Tabelle 6 zeigt)[14] nicht gut ab, dennoch ist GRATIS eins der ersten

[13] Jedenfalls nicht bis zum Zeitpunkt der Implementierung von ScatterClipse.

[14] Das entwickelte Eclipse-Plugin fungiert laut den Entwicklern u.a. als Brücke zu der davor besprochenen Eclipse-basierten TinyDT Umgebung, die ebenfalls an der Vanderbilt Universität entwickelt wurde. Insofern sollten in der unteren Bewertungstabelle der Studie die Merkmale beider Rahmenwerkzeuge eher gemeinsam in Betracht gezogen werden. Denn durch die Eclipse-

generativen modellgetriebenen Werkzeuge zur automatisierten Anwendungsentwicklung für drahtlose Sensorknoten. Mit Hilfe von GRATIS wird, wie die Entwickler beschreiben, lediglich „Glue Code“ generiert, der sich auf die Schnittstellen der TinyOS-Komponenten beschränkt. Dies ist wahrscheinlich auf die eben erwähnte Einschränkung des Codegenerierungsprozesses zurückzuführen. Generierung von Infrastrukturcode wie bei ScatterClipse wird nicht gewährleistet. Ferner bleibt der Test der zu entwickelnden Anwendungen insgesamt ausgeblendet und es werden auch keine Simulations- und Verwaltungsfunktionalitäten in den Entwicklungsprozess integriert.

Eine grafische Entwicklungsumgebung, die ebenfalls die komponentenbasierte Entwicklung für TinyOS unterstützt, ist das von der Cleveland State University entwickelte TOSDev Rahmenwerk. [Cart06] Die komponentenbasierte Modellierung ist im Vergleich zu GRATIS anspruchvoller. So können zwischen den Komponenten vorhandenen Assoziationen semantische Eigenschaften zugewiesen werden, wodurch die Abhängigkeiten der verknüpften Komponenten feinkörniger ausgedrückt werden können. Dies führt dazu, dass der daraus generierte Codeanteil deutlich steigt. Ferner bietet die Modellierung flexible Konfigurationsmöglichkeiten der einzelnen Komponenten, indem beispielsweise beliebig neu definierte Schnittstellen zum Komponentenmodell hinzugefügt werden können. Dies ist u.a. deshalb möglich, weil jedes einzelne Komponentendiagramm intern bearbeitet werden kann, was den Detaildiagrammen der Firmwaremodule bzw. der Testfalldiagramme in ScatterClipse vergleichbar ist: “The wiring editor also allows the developer to dig into any particular component all the way to the lowest level. Opening any component to go to a lower level results either in a wiring diagram or a source file, depending on the component. Such hierarchical views provide a nice cognitive advantage—the views are presented in ‘mind-sized’ chunks, allowing developers the ease of use of small and manageable diagrams, while at the same time carrying all the semantic information that is inherent in the wiring diagrams.” [Cart06]

Ferner bietet das Rahmenwerk weitere unterstützende Funktionalitäten wie einen integrierten Compiler der nesC-Sprache sowie einen Burner zur Vorbereitung der kompilierten Codeartefakte. In dem im Jahr 2006 veröffentlichten Beitrag zu TOSDev wird angekündigt, das Rahmenwerk weiter entwickeln zu wollen: „We are currently working on extending TOSDev to support TinyOS 2.0 and nesC 1.2 (…). TOSDev will provide different visual cues for generic objects (…).“ [Cart06] Ruft man jedoch die offizielle Seite des Projekts auf [WebCart], stellt man fest, dass die letzte Aktualisierung ebenfalls im Jahr 2006 vorgenommen und das Projekt anscheinend eingefroren wurde[15]. Vom Codegenerierungsprozess wird im Paper kaum etwas gesagt. Ferner findet die Modellierung nicht gemäß dem modellgetriebenen Paradigma systematisch auf Basis eines zuvor festgelegten Metamodells zur Komponentenmodellierung statt. Daher kann hinsichtlich der formalen Modellierung nicht von einer Domänensprache gesprochen werden. Somit ist das TOSDev Rahmenwerk lediglich als eine grafische Entwicklungsumgebung anzusehen, die keinerlei Methodik für den Softwareentwicklungsprozesses der Sensoranwendungen aufweist.

Eine grafische Umgebung für TinyOS, die zusätzlich Simulationsfunktionalitäten unterstützt, ist das an der Universität Berkeley entwickelte Werkzeug Viptos (Visual Ptolemy and TinyOS). [Cheo05] Das Besondere an diesem Rahmenwerk ist, dass die erstellten Modelle auf zweierlei Arten verwendet werden können: Einerseits können sie in Quellcode transformiert werden, welcher dann kompiliert und auf den Sensorknoten ausgeführt werden kann. Anderseits können aus den Modellen für die

---

Integration versteht sich GRATIS als die grafische Erweiterung des TinyDT Rahmenwerkes. Dies wurde in der Studie leider nicht berücksichtigt.

[15] Bei Überprüfung des Linkes am 02.08.09 wurde festgestellt, dass der Link nicht mehr existiert.

Simulationsumgebung TOSSIM [Levi03] entsprechende Codeartefakte erzeugt werden, anhand derer die Merkmale des Sensornetzes sowie die einzelnen Sensorknoten näher beobachtet werden. Wie der unteren Vergleichstabelle (vgl. Tabelle 6) der Studie zu entnehmen ist, ist Viptos das einzige Werkzeug, das Simulationsfunktionalitäten bei der grafischen Anwendungserstellung mit einbezieht. Hierzu stützt sich die grafische Darstellung auf das Modellierungsrahmenwerk VisualSense. [Bald04] Dieses Modellierungsrahmenwerk wurde entwickelt, um eine komponentenbasierte Konstruktion solcher Modelle zu unterstützen. Es unterstützt Akteur-orientierte Definitionen von Netzwerkknoten, drahtlose Kommunikationskanäle, physikalische Medien wie akustische Kanäle und drahtgebundene Subsysteme. „This modeling is designed to support a component-based construction of such models. It supports actor-oriented definition of network nodes, wireless communication channels, physical media such as acoustic channels, and wired subsystems." [Bald04]

Daher bieten die grafischen Diagramme umfangreiche Funktionalitäten, um zahlreiche Aspekte adäquat zu repräsentieren. Allerdings ist die Modellierung stark an der Repräsentation des Netzverhaltens orientiert, um daraus die entsprechenden Codeartefakte für den Simulator zu erzeugen. So werden Modellkonstrukte zur Verfügung gestellt, die Kommunikationskanäle, Paketverlust, Energieverbrauch etc. darstellen. Das Verhalten der Anwendungen und deren Softwarearchitektur stehen weniger im Fokus der Modellierung, die daher auch weniger softwaretechnisch motiviert ist. Das Rahmenwerk bleibt lediglich ein grafisches und kein modellgetriebenes Werkzeug in dem Sinne, dass kein strukturierter Entwicklungsprozess bei der Anwendungserstellung zugrunde liegt.[16] Eine Domänensprache auf Basis eines beliebig zu erweiternden Metamodells liegt nicht vor. Dies beschränkt die Offenheit der Architektur bezüglich der Erweiterbarkeit des Rahmenwerkes und dessen Modellierungstechnik hinsichtlich neu auftretender Fragestellungen in der dynamischen Domäne der drahtlosen Sensornetze. Gleiches gilt für die Modellvalidierung um die Robustheit des Entwicklungsprozesses zu steigern. Benutzerdefinierte (semantische) Randbedingungen im Hinblick auf die Plattformeigenschaften werden im Rahmenwerk nicht unterstützt.

Ferner bleibt die Codeerzeugung in ihrer Funktionalität beschränkt. Daher sprechen die Autoren auch von einer Transformation der grafischen Diagramme in Codeartefakte: „Viptos allows developers to create block and arrow diagrams to construct TinyOS programs from any standard library of nesC/TinyOS components. The tool automatically transforms the diagram into a nesC program that can be compiled and downloaded from within the graphical environment onto any TinyOS-supported target hardware." [Cheo05] Eine Weiterentwicklung des Viptos Rahmenwerks hinsichtlich der Unterstützung eines umfassenden Softwareentwicklungsprozesses der Anwendungen der Sensorknoten scheint zum jetzigen Zeitpunkt nicht in Sicht zu sein, da auf der offiziellen Seite des Projektes [Webviptos] die letzte Versionsausgabe und somit Aktualisierung am 9 Februar 2007 vorgenommen wurde.

Tabelle 6 zeigt die vorgestellte Vergleichstabelle der Studie [Teng08], in der auch die Übersicht des Studienresultats vorgestellt wird. Die tabellarische Gegenüberstellung der betrachteten Werkzeuge findet auf Basis der eingangs geschilderten Anforderungen statt, die die Autoren an gegenwärtige Werkzeuge für die Anwendungsentwicklung drahtloser Sensorknoten stellen. Im Folgenden wird bezüglich jeder einzelnen Anforderung dargestellt, inwieweit ScatterClipse diese erfüllt.

---

[16] Ein Rahmenwerk, das hingegen den Softwareentwicklungsprozess modellgetrieben zur automatisierten Generierung von Codeartefakten auf Basis einer Domänensprache für eine Simulationsumgebung drahtloser Sensornetze unterstützt, ist in [Sadi07] realisiert. Dieses wird im weiteren Verlauf des Kapitels behandelt.

Component	Feature	IDE for TinyOS/nesC					
		TinyDT	TinyOSIDE	TOSDev	YETI	Viptos	GRATIS
Modeling						x	x
PManager	Wizard	x	x		x		
	Properties	x	x		x		
Editor	Graphical			x	x	x	x
	Syntax highlighting	x	x	x	x		
	Code navigation	x			x		
	Code completion	x		x	x		
	Code folding			x	x		
	Spell checker				x		
	Syntactic detection				x		
	Error location		x		x		
	Outline				x		
	Internal Parser	x			x		
Compiler		x	x	x	x	x	x
Debugger							
Simulator	Console					x	
	Graphical						
Burner	Serial burning	x	x	x	x	x	
	OTAP						
Monitor							

Tabelle 6 Vergleich der WSN Entwicklungsumgebung nach der Studie von Teng [Teng08]

Modellierung: Dies ist das Herzstück der ScatterClipse-Werkzeugkette, welches durch die generative architekturzentrierte Infrastruktur bestehend aus Visual ScatterUnit und ScatterFactory realisiert wurde.

Projekt Manager: Dank der Eclipse-Integration wird ein Projekt-Manager von jedem Rahmenwerk der Werkzeugkette unterstützt. Ferner besteht ein zentraler Ordner für jedes erstellte drahtlose Sensornetz, in dem sämtliche Konfigurationsdateien abgelegt werden. Dies wird von einem speziellen Wizard realisiert, das vom Root-Plugin des Rahmenwerks ScatterPlug zur Verfügung gestellt wird.

Editor: Sowohl ein textueller als auch ein grafischer Editor sind vorhanden. Der textuelle Editor, welcher vom Entwicklungs-Plugin des ScatterEditor Rahmenwerks angeboten wird, ist für die Codierung der Anwendungen zuständig. Da der Editor auf Basis des Eclipse-Projektes CDT realisiert wurde, werden zahlreiche der in Tabelle 6 aufgelisteten Merkmale standardmäßig unterstützt. Ferner werden die Merkmale von CDT von der Eclipse Gemeinschaft ständig erweitert und optimiert. Der grafische Editor in ScatterClipse ist für die Modellierung zuständig und wird daher von der architekturzentrierten Infrastruktur angeboten. Beim grafischen Editor handelt es sich um einen GMF-generativen Editor, der sich automatisiert aus dem EMF Metamodell generieren lässt.
Ferner liefert er einen Adapter zur dynamischen Einbindung der benutzerdefinierten Randbedingungen um die Live-Validierung der Modelle zu bewerkstelligen.

Simulator: Diese Funktionalität wird lediglich von der Entwicklungsumgebung Viptos unterstützt. In der ScatterClipse-Werkzeugkette wird diese Funktionalität durch den Simulationsdienst von ScatterUnit bewerkstelligt (wie z.B. Simulation der Topologieänderung).

Compiler: Wird durch das Entwicklungs-Plugin von ScatterEditor angeboten und ist auf Basis des standardmäßig in Eclipse integrierten ANT-Projektes realisiert.

Burner: Ein Burner existiert als serielle Variante und über Funk (OTAP). Die serielle Variante wird analog zum zuvor erwähnten Compiler realisiert. Die funkgestützte Softwareausbringung wird durch das in Reitern organisierte Verwaltungs-Plugin unterstützt und wurde dort als eigenständiger Reiter realisiert. Keines der in der Studie untersuchten Werkzeuge unterstützt diese Funktionalität.

Überwachung: Diese Funktionalität wird ebenfalls von keiner Entwicklungsumgebung unterstützt. In ScatterClipse sind unterschiedliche Techniken vorhanden, die je nach Zweck die entsprechende Überwachungsfunktionalität realisieren. So kann der Sensorstatus durch das Visualisierungs-Plugin in ScatterPlug visualisiert und überwacht werden. Ferner gewährt das Statistik-Plugin von ScatterPlug eine statistische Analyse der gesammelten Sensordaten, um Rückschlüsse auf die Sensorik sowie die Sensorumgebung zu ziehen.

Debugging: Ein visueller, automatisierter Test- und Debuggingprozess wurde in ScatterClipse mittels Visual ScatterUnit realisiert. Diese Funktionalität wird ebenfalls von keiner der berücksichtigten Entwicklungsumgebungen angeboten.

Die ScatterClipse Werkzeugkette erfüllt damit alle Ansprüche, die in dieser Studie an WSN-Entwicklungswerkzeuge gestellt werden.

Wie bereits erwähnt, gibt es bislang keine Entwicklungsumgebung, die den Test der zu entwickelnden Anwendung während ihrer Erstellung softwaretechnisch unterstützt. Jedoch existieren mehrere Versuche, bzw. Werkzeuge, die sich mit dem Test drahtloser Sensorknoten auf Basis unterschiedlicher Ansätze beschäftigen. Im Folgenden werden diejenigen Ansätze besprochen, die insbesondere für die Testautomatisierungsschicht ScatterUnit und ihre modellgetriebene Ergänzung Visual ScattterUnit von Relevanz sind.

Da das robuste Routing in drahtlosen Sensornetzen aufgrund ihrer Betriebsbedingungen eine anspruchsvolle Herausforderung ist, widmen sich bestimmte Ansätze hinsichtlich des Tests hauptsächlich dieser Problemstellung. Etliche Simulatoren für drahtlose Sensornetze [WebWSN-Simu] sind speziell zu diesem Zweck entwickelt worden (z.B. J-Sim [Sobe05], Avrory [Titz05], ns-2 [Witt06]). Dabei liegt der Fokus überwiegend darauf, wie sich das Netz bzw. die im Netz agierenden Sensorknoten bei der Ausführung des Routing-Protokolls verhalten. Zwar werden dadurch diverse Aspekte überprüft, was den Einsatz von Simulatoren unverzichtbar macht, jedoch bleibt die Anforderung eines Tests unter realen Bedingungen bei Simulationswerkzeugen unerfüllt. Ferner gerät oft die Untersuchung der Anwendung selbst, die das Protokoll realisiert, in den Hintergrund.

So wurde stattdessen der Versuch unternommen, Routing-Protokolle realitätsnäher zu testen, z.B. indem fahrende Fahrzeuge zum Zweck des Tests der Mobilität der Sensorknoten im Netz eingesetzt wurden. Ein solcher Versuch [Malt00] wurde von einer Forschergruppe der Universität Carnegie Mellon unternommen um das Dynamic Source Routing Protokoll (DSR) [Broc99] [Perk01] zu testen. Hierzu wurde zum Testzweck ein drahtloses Sensornetz erstellt, wobei zwischen den Senderknoten und dem Empfänger fünf Zwischenknoten gesetzt wurden. Während sowohl der Sender- als auch der Empfängerknoten ihre Position nicht änderten, wurden die fünf Zwischenknoten jeweils in einem fahrenden Fahrzeug platziert. Das Fahrverhalten jedes einzelnen Fahrzeugs wurde der jeweiligen Verkehrslage spontan angepasst, wodurch das Bewegungsmuster jedes Wagens und des sich in ihm befindlichen Sensorknotens nicht im Voraus bestimmt

werden konnte. So variierte die Fahrgeschwindigkeit der einzelnen Wagen zwischen 25 und 40 km/h. “The road we use is open to general vehicle traffic and has several Stop signs, so the speed of each node varies in a complex fashion, just as it would in any real network.“ [Malt00] Das versendete Paket soll nun anhand des Routing-Protokolls seinen Weg unter real nicht vorhersehbaren Testbedingungen zum Empfänger finden. „With the vehicles, radios, and site used in our testbed, we forced the protocols to operate in an environment in which all links between nodes change status at least every 220 seconds. Ignoring the additional factor of packet loss due to wireless errors, on average, the network topology changed every 4 seconds.“ [Malt00] Die Experimente wurden mehrfach wiederholt, wobei jedes Experiment zwischen 30 und 120 Minuten dauerte.

Ein solches Szenario bietet zwar vorzügliche realitätsnahe Testbedingungen für das zu testende Protokoll, insbesondere hinsichtlich der Mobilität der Sensorknoten im Netz, die eine besondere Herausforderung an solche Routing-Protokolle darstellt, jedoch scheitert ein solcher Ansatz schnell an der Anforderung der Skalierbarkeit. Mit anderen Worten, ein solcher Ansatz stößt an seine Grenze, wenn es um die Frage geht, wie ein Protokoll für ein Netz mit einer großen Anzahl von agierenden Sensorknoten zu testen ist. Hier sind simulative Eingriffe zur automatisierten Testdurchführung unerlässlich. Die Integration simulativer Funktionalitäten trägt zu einer deutlichen Erhöhung des Automatisierungsgrads vom „real“ auf den Sensorknoten auszuführenden Testfall bei. Daher prägt dieser hybride Ansatz den Entwurf von ScatterUnit, der Testautomatisierungsschicht der Werkzeugkette. Dies sieht man z.B. im Einsatz des Simulationsdienstes in ScatterUnit zur dynamischen Konfiguration der Netztopologie während der Testdurchführung.

Ein Testwerkzeug, das ebenfalls diesen Ansatz verfolgt, ist SeNeTs [Blum04], welches an der Universität Rostock entwickelt wurde. Eine Basisstation sendet für die Steuerung des Testverlaufs entsprechende Kontrollbefehle zu den Knoten im Netzwerk (z.B. um die Versendung des Datenpakets zu initiieren) und die Knoten senden Informationen über alle relevanten Ereignisse zurück an die Basisstation. Diese Informationen entstehen durch die Protokollierung des Testverlaufs von jedem im Netz beteiligten Sensorknoten. Hierzu bedienen sich die Sensorknoten eines separaten Kommunikationskanals, um die Test-(Meta)daten zu verschicken: “The secondary communication channel is an administration channel only used by SeNeTs components. This channel transmits controlling and logging messages. It is independent of the primary communication channel and uses a different communication method, e.g. Ethernet or Ultrasound.” [Blum04] Drahtgebundene Kommunikation zur Übertragung der Testinformationen beschränkt jedoch die Realitätsnähe der Testdurchführung. Der Einsatz einer zusätzlichen Kommunikationsmethode speziell zum Zweck der Übertragung der Testdaten erfordert die Bestückung jedes einzelnen Sensorknotens mit einem entsprechenden Funkmodul. Ferner werden die während des Voranschreitens der Testdurchführung von den einzelnen Knoten ständig versendeten Nachrichten von auf der Basisstation laufenden Modulen analysiert, um Rückschlüsse hinsichtlich des Verhaltens der zu testende Anwendung ziehen zu können. Demgemäß wird ein zentralistischer Ansatz bei der Protokollierung und Evaluierung der Testinformationen verfolgt, was in der sensitiven Domäne drahtloser Sensornetze schnell zum Engpass wird. Denn die SeNeTs Testinfrastruktur wurde allgemein für Ad-hoc Netze entworfen, bei denen die Ressourcen im Vergleich zu drahtlosen Sensornetzen weniger beschränkt sind. Bei drahtlosen Sensornetzen ist hingegen zu beobachten, dass die Übertragung eines Pakets bei einem Sensorknoten durchaus fehlschlagen kann, wenn der Funkkanal zu oft von einem anderen Knoten belegt ist. Ein Grund könnte die dadurch verursachte häufige Kollision auf dem Funkkanal sein. Daher ist ein solcher ressourcenintensiver, zentralistischer Kommunikationsmechanismus nicht optimal geeignet.

Aus diesem Grund wurde ein dezentraler Steuerungsmechanismus zur Ausführung des Testverlaufs in der Testschicht ScatterUnit verfolgt, um diesen unerwünschten

Effekt möglichst minimal zu halten. Dies wurde, wie erläutert, vor allem dadurch erreicht, dass die durch *local logging* protokollierten Daten erst nach dem Ende der verteilten Testdurchführung an die auf der Infrastruktur laufenden Prüfroutinen versendet werden. Ferner werden die Kontrollbefehle zur Koordination des verteilten Testverlaufs zusammen mit den Codeartefakten des Testfalls einmalig in der Form von Testskripten unmittelbar vor dem Start der Testausführung auf die jeweiligen Sensorknoten ausgebracht. Was das SeNeTs Werkzeug anbelangt, so gilt zudem, dass selbst die Codierung von Testfällen auf Basis des Testwerkzeugs ein schwieriges Unterfangen bleibt. Zwar wird die Durchführung des Tests durch das Werkzeug automatisiert bewerkstelligt, jedoch müssen die dazugehörigen Testfälle manuell vom Benutzer entwickelt werden. Hierzu muss sich der Entwickler mit verschiedenen technischen Belangen des Tests befassen, um richtige Testfälle zu entwickeln, wodurch eine robuste Testdurchführung gewährleistet werden kann. Diese recht komplexe und fehleranfällige Tätigkeit wird in ScatterUnit dem Benutzer abgenommen und durch die modellgetriebene Entwicklung der Testfallmodelle an den Codegenerator delegiert. Dies wurde mittels der Ergänzung Visual ScatterUnit erreicht, wozu bislang offenbar kein Pendant in der WSN Domäne existiert. Es finden sich jedoch verwandte Testwerkzeuge hinsichtlich der Verwendung von Aktivitätsdiagrammen bei der Modellierung der Testfälle.

Ein Vertreter ist das in [Lenz07] vorgestellte Werkzeug. Lenz et al. entwickelten ein modellgetriebenes Werkzeug, das die Modellierung von Testfällen für Anwendungen ermöglicht, die gemäß der dienstorientierten Architektur (SOA) entwickelt wurden. Die Modellierung der Testfälle stützt sich – ähnlich wie in ScatterClipse – auf den Einsatz von UML-basierten Aktivitätsdiagrammen. Dies geschieht mit Hilfe des UML Testing Profiles[17] [WebTestProfile]. Die Schlüsselidee des Ansatzes ist die Entwicklung von Testfallmodellen unabhängig von der Technologie, die für die Dienstaufrufe benutzt wird. Zu diesem Zweck enthalten die Sequenz-Diagramme technologieunabhängige Methodenaufrufe, wobei diese Methoden in einem UML-Klassendiagramm des zu testenden Systems dargestellt werden. Während der Ausführung des Tests ruft der generierte Testfall die modellierten Methoden auf, die durch eine vom Modell des getesteten Systems generierte Adapter-Klasse implementiert werden. In diese systemabhängigen Adapter-Klassen muss der Benutzer den adäquaten Code für das jeweilige System einmalig manuell implementieren. Somit fungieren die UML-basierten Aktivitätsdiagramme zum Einen als Abstraktionsmechanismus zur plattformunabhängigen Testdurchführung für SOA-gestützte Anwendungen und zum Anderen wird die verteilte technologieunabhängige Testdurchführung koordiniert.

Betrachtet man die Spezifikation des UML-Testing Profiles, so handelt es sich um ein 113 Seiten umfassendes Papier, dessen Handhabung ein recht komplexes Unterfangen ist. Oft werden nicht alle im Profil angebotenen Eigenschaften zum unterliegenden Testszenario benötigt. Nichts desto trotz muss man sich zunächst mit dem gesamten Profil befassen, um zu den relevanten Aspekten zu gelangen, die verwendet werden sollen. Darüber hinaus haben die recht komplexen UML-basierten Modelle gegenüber einer domänenspezifischen Sprache (DSL), wie sie in Visual ScatterUnit eingesetzt wird, den Nachteil, dass sie nicht über ein so ausführliches Wissen der Domäne verfügen, wie die in DSL erzeugten Modelle. Da die Modellelemente der DSL für konkrete architektonische Konzepte oder Aspekte der Domäne stehen, bietet ein in DSL dargestelltes Modell einen höheren Abstraktionsgrad und ist zugleich konkret. Die semantische Lücke zwischen Modell und Code wird auf diese Weise kleiner. Als Nebeneffekt vereinfacht dies die Umwandlung der Modelle in Code, da die schrittweise Transformation der Modelle zu Code oft übersprungen werden kann, weil die unterliegende Plattform bekannt und eindeutig eingegrenzt ist.

[17] Siehe Glossar.

Eine Testsprache, die auch über eine breite Unterstützung der Toollandschaft verfügt, ist TTCN (Testing and Test Control Notation) [WebTTCN]. Diese liegt aktuell in der dritten Version vor, weshalb oft von TTCN-3 gesprochen wird. Während sich TTCN bislang vor allem auf den Test von Kommunikationsprotokollen beschränkte, unterstützt TTCN-3 den Softwaretest von Sensoren und Aktuatoren, die über Bussysteme angeschlossen werden. TTCN-3 sieht drei Techniken vor, die gemeinsam zur Modellierung der Testfälle zum Einsatz kommen. So werden die Testfälle textuell, tabellarisch und grafisch je nach Fragestellung spezifiziert. TTCN-3 wird von der *International Telecommunication Union* (ITU) [WebITU] betreut und es werden Zertifizierungsprogramme für den Einsatz angeboten. Der Bedarf an Zertifizierungsmaßnahmen verweist allerdings auf einen erheblichen Grad der Modellierungskomplexität, was eine hohe Lernkurve zur Folge hat und somit die Handhabung der Modellierungsprache erschwert. Dies ist sicherlich auf die vielfältigen und unterschiedlichen Modellierungstechniken zurück zu führen. Die Integration modellgetriebener Elemente in den Testprozess, wie sie in ScatterClipse vorhanden sind, wäre hier eine sehr interessante Herausforderung. Ansätze in diese Richtung existieren bereits [Zand05]. Ferner würde die Unterstützung eines kognitiven Testprozesses, bei dem die Testfälle iterativ auf Basis der Hypothesen der Tester verfeinert werden (wie bei Visual ScatterUnit), die Beherrschbarkeit sowie Präzision der Erstellung der Testfallmodelle deutlich erhöhen. Ein kognitiver modellgetriebener Testprozess findet anscheinend bislang in TTC-3 keine Unterstützung. Alles in allem bietet die Modellierungssprache mächtige Eigenschaften zur Unterstützung des Testprozesses, jedoch stehen dabei Feldbussysteme und verteilte Infrastrukturen und nicht Ad-hoc-Netze als Testdomänen im Fokus.

Ein weiteres modellgetriebenes Werkzeug, das auf einer Aktivitätsdiagramm-ähnlichen Modellierungssprache beruht, wird von Ulbrich et al. vorgestellt [Ulbr05a, Ulbr05b], wobei die Autoren jedoch nicht erläutern, warum eine eigene Notation in der Modellierungssprache verwendet wird. Jedenfalls wird im Beitrag eine modellgetriebene visuelle Sprache nach dem MDA-Ansatz für Sensornetze vorgestellt. Das Hauptziel der Sprache ist die Abstraktion von spezifischen Endgeräten sowie die Verdeckung der Verteilung beim Einsatz dieser Geräte. Dementsprechend besteht eine nach der Modellierungssprache repräsentierte Anwendung aus einer oder mehreren Prozessen, wobei ein Prozess einen Fluss von im Netz stattgefundenen Aktionen bzw. Ereignissen darstellt. Hiermit verfügt der Verlauf eines Prozesses – analog zu den Aktivitätsdiagrammen – über einen Start- und einen Endknoten. Prozesse können dabei parallel oder sequenziell ablaufen. Ferner unterstützt die Modellierungssprache die Gruppierung einzelner Ereignisse bzw. Aktionen, um logische Zusammenhänge im Netz zu modellieren (z. B. geografische Lage, Zugehörigkeit der betroffenen Endgeräte zu einer Person, etc.).

In dem Beitrag wird zwar ein modellgetriebener Prozess zur Anwendungsentwicklung von Sensorknoten propagiert, die Autoren betrachten jedoch lediglich den Aufbau der Modellierungssprachen und deren Eigenschaften. Weitere, für den modellgetriebenen Entwicklungsprozess essentielle Bausteine wie die Metamodellierung, die Validierung der Modellierungssprache und der Codegenerierungsprozess werden nicht berücksichtigt. Die Modellierungssprache sollte jedoch im Kontext des modellgetriebenen Prozess dargestellt und nicht isoliert, sondern zusammen mit den eben erwähnten Komponenten betrachtet werden. Denn erst dadurch kann eine adäquate Beurteilung hinsichtlich der Realisierbarkeit und Einsatztauglichkeit der vorgestellten Modellierungssprache getroffen werden.

Der einzige Aspekt des modellgetriebenen Prozesses, der neben der Modellierungssprache angesprochen wird, ist die M2M-Transformation, durch die das PSM aus dem PIM gewonnen wird. Dies wird in einem Unterkapitel erläutert, wobei sich die Darstellung auf eine theoretische Grundlage beschränkt und mehr die

Konzeption der M2M-Transformation geschildert wird. Ebenso wenig wird darauf eingegangen, wie die Realisierung des Transformationskonzepts vonstatten geht. Alles in allem handelt es sich hier eher um einen Vorschlag, in dem Eigenschaften und Entwurf einer zu entwickelnden Modellierungssprache entworfen werden. Nachfolgende Arbeiten zu dem vorgestellten Beitrag existieren bislang anscheinend nicht, was die Realisierbarkeit des Ansatzes überhaupt in Frage stellt.

Ferner existieren modellgetriebene Werkzeuge zur Generierung von Softwareartefakten zur automatisierten Erstellung von Anwendungen drahtloser Sensornetze. So wird in [Losi07] ein modellgetriebenes Rahmenwerk zur automatisierten Generierung von Softwareartefakten für eine plattformunabhängige Erstellung von Anwendungen drahtloser Sensorknoten vorgestellt. Dies soll mittels einer universellen Modellierungssprache bewerkstelligt werden, in der allgemein gebräuchliche Aspekte der WSN-Domäne dargestellt werden. Für die Modellierungssprache wird UML verwendet. Damit wird in der Arbeit eindeutig ein MDA-Ansatz verfolgt. Für die Definition der Modellierungssprache wurde – ähnlich wie bei ScatterClipse – ein EMOF-Metamodell mit Hilfe des EMF-Rahmenwerks entwickelt. Allerdings werden Verhaltensdiagramme (wie z.B. Aktivitätsdiagramme) bislang nicht unterstützt, was sicherlich einen gravierenden Mangel in der Ausdrucksmächtigkeit der Modellierungssprache darstellt. Um die bereits angesprochene semantische Lücke zu reduzieren wird für jede Zielplattform ein zugehöriges, plattformspezifisches Modell (PSM) zur Verfügung gestellt, in welches das jeweilige Modell der universellen Modellierungssprache transformiert wird. Zur Unterstützung jeder unterliegenden Plattform bedarf es daher einer entsprechenden Model-zu-Model-Transformation (M2M). Dabei werden die Transformationsvorschriften mit Hilfe der deklarativen imperativen ATL Transformationssprache [WebATL] spezifiziert, welche es ermöglicht, Zuordnungen zwischen verschiedenen Modellkonstrukten festzulegen. Aus dem jeweiligen PSM wird anschließend mit Hilfe des MOFScript-Plugins von Eclipse der zugehörige Code generiert. Leider wird in dem Beitrag nicht genauer darauf eingegangen, wie die Codegenerierung mit Hilfe des Plugins vonstatten geht.

Zwar soll der Beitrag der Arbeit eine universelle Modellierungssprache für WSN sein, die Phasen des modellgetriebenen Prozesses werden jedoch anhand eines Beispiels für eine Plattform, nämlich TinyOS, illustriert. Es wird nicht gezeigt, wie aus demselben (universellen) Modell unterschiedliche Codeartefakte für deren jeweilige Plattformen generiert werden, obwohl dieser Aspekt die Arbeit von anderen abgrenzen soll. Die Beschränkung auf TinyOS dominiert die Vorstellung der Konzepte des Rahmenwerks, was die Realisierbarkeit einer universellen Modellierungssprache kaum nachweist. Es handelt sich bei der Arbeit eher um eine Anregung bzw. einen Ansatz, der von seiner Realisierung noch weit entfernt ist. Des Weiteren werden im vorgestellten Ansatz zwei Kernaspekte nicht unterstützt: Verhaltensdiagramme in der vorgeschlagenen universellen Modellierungssprache werden nicht in Betracht gezogen, wodurch sich die Modellierung auf statische Aspekte der Anwendung beschränkt. Das Verhalten der modellierten Anwendungen wird somit außer acht gelassen. Ein Testprozess ist damit nicht im Blickfeld. Andererseits wird in der gesamten Arbeit keine Modellvalidierung erwähnt, was jedoch für eine robuste modellgetriebene Anwendungsentwicklung unerlässlich ist. Nachfolgende Entwicklungen dieser Arbeit sind bislang nicht zu finden.

Von Sadilek et al. wird ein Forschungsvorhaben vorgestellt, in dem die Eigenschaften einer domänenspezifischen Sprache für die modellgetriebene Entwicklung von WSN-Anwendungen im Mittelpunkt steht. [Sadi07, Sadi08] Zwar ist inzwischen die Zielsetzung des Vorhabens vollständig neu definiert worden,[18] was möglicherweise

[18] Das Forschungsprojekt beschäftigt sich inzwischen mit der strukturierten Entwicklung und Wartung, von Metamodellen und deren Domänensprachen im Hinblick der in den Modellen benötigten operationalen Semantik. Mehr dazu in [WebSadilek]

darauf hinweist, dass der vorgestellte Ansatz keine entsprechende Umsetzung gefunden hat. Nichts desto trotz soll hier auf dessen Konzept im Vergleich zu ScatterClipse eingegangen werden. Die Schlüsselidee liegt in einer Domänensprache für drahtlose Sensornetze, die nicht nur auf der Basis von Metamodellen eingesetzt wird, sondern auch die Technik der Metaprogrammierung verwendet. Bei Metaprogrammiersprachen handelt es sich im Allgemeinen um Sprachen, die das Verhalten fremder Programme, welche in der Regel in anderen Programmierparadigmen geschrieben sind, steuern und manipulieren können. Dies ist möglich, weil Metaprogramme in der Lage sind, die zu steuernden Fremdprogramme (auch Fremdobjekte genannt) als eigene Datenstrukturen zu handhaben. [WebMetaprogrammierung]

Die Autoren begründen ihren Ansatz der Verwendung von Metaprogrammierung neben Metamodellen, indem sie anmerken, dass letzteren eine Beschreibung der operationalen Semantik hinsichtlich des Verhaltens der zu modellierenden Domäne fehlt: „Metamodels miss inherent operational semantics. They only contain a description of the language's abstract syntax, i.e. which language concepts are available and how they can be combined. They miss a description of the language's operational semantics, i.e. how the language concepts should behave when executed. (…) We can conclude that metamodels make the language prototyping cycle as we expect it for WSNs expensive because they miss inherent operational semantics.“ Dieser Argumentation steht bei ScatterClipse entgegen, dass die operationale Semantik der Domänensprache vollständig durch die zentral abgelegten Checks-Ausdrücke definiert wird. Wie bereits in 2.5 geschildert, kann man auf die semantischen Ausdrücke mit dem GMF-Adapter dynamisch zur Modellierungszeit zugreifen und sie interpretieren. Demgemäß können bei jedem Iterationszyklus flexibel und ohne weiteren Aufwand während der (Weiter)Entwicklung der Domänensprache entsprechende Checks-Ausdrücke verfeinert bzw. neu definiert werden.

Ferner sind die Autoren der Ansicht, dass die automatisiert aus der Domänensprache generierten Artefakte nicht direkt auf den Sensorknoten auszuführen sind, da dies nur schwer möglich sei. Daher schlagen sie vor, die Artefakte lediglich auf einer Simulationsumgebung auszuführen „deploying and testing programs on the target platform is very costly. This means that it's not feasible to test programs written in the DSL directly on the target platform especially in the early rounds of the prototyping cycle when the DSL is used more for the purpose of evaluating the DSL itself than of developing the system. Therefore, it is necessary that the DSL can be simulated.” [Sadi07]

Die Entwicklung der Domänensprache auf Basis einer Simulationsumgebung um dann die DSL erst nach ihrer Entwicklung direkt auf den Sensorknoten einzusetzen, entspricht nicht der Anforderung der realitätsnahen Entwicklung einer domänenspezifischen Modellierungssprache. Die Argumentation, dass die Ausbringung und der Test der Sensorsoftware kostspielig seien, trifft auf ScatterClipse nicht zu. Denn mit Hilfe von Visual ScatterUnit und den Verwaltungsfunktionalitäten der Werkzeugkette können Testfälle auf die jeweiligen Sensorknoten nahtlos ausgebracht und dort ausgeführt werden.

Von Wada et al. wird ein modellgetriebenes Rahmenwerk zur Entwicklung von WSN-Anwendungen vorgestellt. [Wada07] Die Modellierung der Anwendungen fußt (ähnlich wie in [Lenz07]) auf der UML-Profiles Technik. Um die formalen UML-Modelle validieren und ausführen zu können, wurde speziell für diesen Zweck eine virtuelle Maschine (Matilda) als wichtige Komponente des Rahmenwerkes entwickelt. Die Modellierungssprache verwendet verschiedene Arten von UML-Diagrammen: Klassen-, Aktivitäts- und Instanzdiagramme. „The proposed UML profile specifies the conventions to build UML models for event detection, WSN applications, and defines stereotypes and tagged-values to precisely describe computationally-complete input models. In the

proposed UML profile, an input model is defined as a set of UML 2.0 class diagrams, sequence diagrams and an instance diagram.“ [Wada07]

Auch hier gelten die bezüglich [Lenz07] angesprochenen Nachteile der Modellierung mit UML-Profiles, insbesondere was die Komplexität bzw. Skalierbarkeit der Modellierungssprache betrifft. So sieht die im Rahmenwerk vorgestellte Modellierungssprache vor, alle im WSN vorkommenden Sensorknoten als eigenständige Instanz im Instanzdiagramm zu modellieren, wobei die Assoziationen zwischen den einzelnen Sensorknoten bzw. Instanzen spezifiziert werden müssen. Dies zeigt sich anhand des Instanzdiagramms für das Anwendungsszenario, welches von den Autoren zur Darstellung der Modellierungssprache vorgestellt wird. Es reichen lediglich zehn Sensorknoten im Anwendungsbeispiel, um die Lesbarkeit des dazugehörigen Instanzdiagramms so gut wie unmöglich zu machen. Umso komplexer dürfte seine Erstellung sein. Dieses Paradebeispiel macht deutlich, wie schnell UML-basierte Diagramme zu einer hohen Komplexität neigen, was ihre Handhabung und Modellierbarkeit erschwert. An dieser Stelle zeigt sich die Stärke des maßgeschneiderten Entwurfs der DSL in ScatterClipse, welche beispielsweise die Sensorknoten mittels deren IDs über die entsprechenden Assoziationen den jeweiligen generischen Behälterdiagrammen „Knotenlaufzeitkonfigurationen NodeA und NodeB“ zuweist (vgl. Abbildung 47 in 5.6).

9.1. Zusammenfassung

Da die ScatterClipse-Werkzeugkette umfangreiche und vielfältige Funktionalitäten anbietet, für deren Einsatz jeweils unterschiedliche Techniken und Methoden verwendet werden, wurden die verwandten Arbeiten gemäß ihrem jeweiligen Einsatzzweck untergliedert und von unterschiedlichen Blickwinkeln betrachtet.

Weil die ScatterClipse-Werkzeugkette im Großen und Ganzen der Anwendungsentwicklung für drahtlose Sensornetze dient, wurden zunächst allgemein Entwicklungswerkzeuge bzw. Umgebungen für drahtlose Sensornetze in Betracht gezogen. Dabei lag die Studie von Teng et. al zugrunde [Teng08], welche Ende Oktober 2008 auf ICSNC 08 vorgestellt wurde (der gleichen Konferenz, auf der auch große Teile der ScatterClipse-Werkzeugkette vorgestellt wurden). In dieser Studie wurden aktuell in der Forschung vorhandene, repräsentative Werkzeuge und Umgebungen zur Entwicklung von Softwareanwendungen für drahtlose Sensorknoten präsentiert und die jeweiligen Ansätze verglichen. Hierzu wurden zunächst die an solche Werkzeuge zu stellenden Anforderungen formuliert, um auf dieser Basis einen Vergleich zwischen den Werkzeugen bzw. Ansätzen vornehmen zu können. Anhand der in der Studie vorgenommenen Untersuchungen wurde klar, dass die Entwicklungswerkzeuge den Testprozess der auf den Knoten laufenden Anwendungen so gut wie ausblenden, was die Robustheit des gesamten Entwicklungsprozesses einschränkt. Ferner zeigte die Studie, dass Überwachungsfunktionalitäten während der Anwendungsentwicklung in den Werkzeugen kaum unterstützt werden. Beides, der automatisierte Test sowie die Überwachung, werden von ScatterClipse in den modellgetriebenen Softwareentwicklungsprozess nahtlos intergriert.

Danach wurden verwandte Lösungsansätze diskutiert, die sich dem Gebiet der Anwendungstests widmen. Diese Beiträge waren anhand von zwei Gesichtspunkten in Bezug auf die ScatterClipse-Werkzeugkette zu untergliedern: Einerseits werden vorhandene Lösungen zum Softwaretest drahtloser Sensorknoten besprochen, andererseits sind modellgetriebene Testwerkzeuge unter die Lupe zu nehmen, bei denen der Fokus auf dem Aspekt der Modellierung und Automatisierung liegt. Hierbei wurde die schnell wachsende Komplexität der eingesetzten Modellierungstechniken festgestellt, welche einen Engpass darstellt. Es wurde deutlich gezeigt, welchen Vorteil eine „purpose-built“ Domänensprache mit sich bringt.

Ferner wurden einige modellgetriebene Werkzeuge für drahtlose Sensornetze besprochen, die sich lediglich mit der Anwendungserstellung beschäftigen, ohne den Testprozess solcher Anwendungen in Betracht zu ziehen. Solche Arbeiten wurden abschließend vorgestellt und diskutiert, wobei deutlich wurde, inwiefern ScatterClipse sich von diesen unterscheidet.

Alle vorgestellten WSN-Werkzeuge haben eines gemeinsam: Die Integration eines automatisierten Testprozesses in den Prozess der Anwendungsentwicklung wird nicht unterstützt. Dies ist ein prägnantes Merkmal der ScatterClipse Werkzeugkette, mit dem sie sich von anderen abgrenzt.

10. Fazit und Zusammenführung

Mit zunehmender Vielfalt und Komplexität des Einsatzes drahtloser Sensornetze steigen die Anforderungen an die auf den Sensorknoten laufende Software. Je anspruchsvoller die Verwendung solcher ressourcenschwacher Netze wird, desto robuster, zuverlässiger und effizienter muss die Entwicklung ihrer Softwareanwendungen vonstatten gehen. Da bislang der Fokus der Forschung meist auf den Hardwareeigenschaften und dem Verhalten des Netzes bezüglich bestimmter Fragestellungen (Energieverbrauch u.a.) liegt, wurde die Anwendungsentwicklung für solche Netze nicht ausreichend berücksichtigt. Dabei besteht selbst für ein sehr energiebewusstes Routingprotokoll weiterhin eine Lücke, wenn dessen Software nicht mit entsprechender Leistungsfähigkeit, Zuverlässigkeit und Realitätsnähe erstellt und getestet werden kann.

Die Optimierung der Anwendungsentwicklung drahtloser Sensornetze ist umso unentbehrlicher, wenn man bedenkt, dass ein solches Netz meist mit einer großen Anzahl von Sensorknoten betrieben wird, die gemeinsam den angestrebten Einsatzzweck des Netzes erreichen sollen. Die verschiedenen Knoten übernehmen im Netz unterschiedliche Rollen, woraus resultiert, dass für die Knoten eines Netzes vielfältige Anwendungen zu entwickeln sind. Die Programmierung solcher Sensornetze muss daher mit einer Produktivität erfolgen, die zugleich ihren Betriebsbedingungen angepasst ist. Ein adäquater Softwareentwicklungsprozess muss diesen Bedingungen Rechnung tragen und zudem einen hohen Grad an Agilität aufweisen. Dies ist das Ziel der in der vorliegenden Arbeit vorgestellten ScatterClipse-Werkzeugkette.

Vor diesem Hintergrund ist der Entwurf von ScatterClipse von der Kombination dreier wesentlicher Eigenschaften geprägt, nämlich der *Produktivität* und der *Robustheit* der Programmierung der Sensorknoten, sowie der *Nähe zu den Einsatzgegebenheiten* der Sensorknoten. Die Randbedingungen, denen der Einsatz von Software auf einem Sensornetz sowie dessen Betrieb unterliegen, wirken diesen drei Aspekten entgegen.

Der Forderung nach Produktivitätssteigerung bei der Erstellung der Softwareanwendungen, die sich aus der Vielfalt der laufenden Anwendungen und dem Massenbetrieb der Sensorknoten ergibt, wird durch einen höheren Grad an Automatisierung begegnet. Die Erstellung von Anwendungen erfolgt daher im Kontext von wiederverwendbaren Softwarefamilien. Hierzu wurde das architekturzentrierte ScatterFactory(2)-Rahmenwerk entwickelt, mit dessen Hilfe die Anwendungen der Sensorknoten aus ihren formalen Modellen generiert werden. Durch den hohen Grad an Automatisierung beschleunigt sich die Laufzeitkonfiguration der zu realisierenden Anwendungen. Ferner können mehrere Anwendungen im gleichen formalen Diagramm modelliert werden, wobei deren gemeinsam verwendete Softwaremodule lediglich einmal modelliert werden müssen. Zudem kann eine modellierte Anwendung einer Vielzahl von Sensorknoten gleichzeitig zugewiesen werden, so dass die generierten Softwareartefakte anschließend automatisiert im Einsatzort ausgebracht werden können. Des Weiteren steigern sich Ersetzbarkeit und Wiederverwendbarkeit der Softwareartefakte, da diese durch die zugehörigen Modellkonstrukte formal repräsentiert werden. Die modellgetriebene Codegenerierung dient zudem der maßgeschneiderten Codeerzeugung, so dass nur so viel Code für die Sensorknoten erzeugt wird, wie für ihre jeweilige Rolle erforderlich ist.

Die Automatisierung erstreckt sich darüber hinaus nicht nur auf die Anwendungserstellung sondern auch auf die Testdurchführung hinsichtlich der Versagensprüfung sowie der Fehlerlokalisierung im Code der zu erstellenden Anwendung. Gerade die steigende Komplexität des Einsatzes drahtloser Sensornetze erfordert eine robustere Erstellung der Softwareartefakte, die sich adäquat testen lassen müssen. Um diesen Anforderungen zu begegnen, wurde zunächst die

Testautomatisierungsschicht ScatterUnit entwickelt, welche den verteilten Testverlauf auf Sensorebene automatisiert durchführt. Zu diesem Zweck wurde ein dezentralisierter Ansatz verfolgt, um mittels *local logging* den Testverlauf zu protokollieren, damit dieser erst nach Beendigung der Testausführung mittels spezieller Prüfroutinen evaluiert und auf Fehler überprüft wird. Hierzu müssen Testskripte implementiert werden, die auf den verschiedenen Sensorknoten des Netzes nebenläufig ausgeführt werden, um die dezentrale Testausführung zu koordinieren und zu steuern. Solche Funktionalitäten werden durch die verschiedenen Dienste ScatterUnits automatisch zur Verfügung gestellt. Damit müssen Dienste, die für verschiedene Testfälle mehrfach verwendet werden, nicht erneut für jeden Testfall implementiert werden. Als Resultat wird die Entwicklung eines auf Basis von ScatterUnit erstellten Testfalls wesentlich effizienter. Dies ist auf den Entwurf der Architektur von ScatterUnit zurückzuführen, welcher zwei prägnante Charakteristika aufweist: Erstens läuft ein Teil des Testcodes direkt auf den Sensorknoten um die verteilte Testausführung zu koordinieren, zweitens findet die Fehlerüberprüfung mittels der Prüfroutinen erst nach Beendigung des Testdurchlaufs statt, so dass die relevanten Aktionen bzw. Ereignisse des Tests nicht während der Testausführung, sondern erst danach von den auf dem Rechner laufenden Prüfroutinen analysiert werden. Diese erwähnten Charakteristika sind unentbehrlich für eine schonende Beanspruchung von Ressourcen im Sensornetz. Damit wird eine Testdurchführung möglich, welche die zu testende Anwendung äußerst geringfügig berührt, so dass das tatsächliche Verhalten der Anwendung unbeeinflusst bleibt. Andernfalls würden die Testergebnisse das Verhalten der getesteten Anwendung unter realen Bedingungen nicht reflektieren. Auf diese Weise können präzisere Hypothesen hinsichtlich der Fehlerursache getroffen werden.

Allerdings bleibt die eigentliche Codierung der Testskripte zur Koordination der verteilten Testdurchführung ein schwieriges Unterfangen. Auf der Ebene von ScatterUnit ist die Implementierung dieses Codes eine komplexe und zeitaufwändige Aktivität: Die Befehle des Testfalls müssen von diversen Testskripten implementiert werden, die dann verteilt auf den verschiedenen Sensorknoten ausgeführt werden. Um die richtige Reihenfolge der Befehle beizubehalten, muss die verteilte Ausführung der Testskripte koordiniert werden. Anschließend muss der Benutzer nach der Testdurchführung die geloggten Testergebnisse manuell im textuellen Protokoll überprüfen, um seine Hypothesen hinsichtlich der Fehlerursache treffen zu können. Es liegt nun auf der Hand, die Automatisierung weiter zu treiben, indem solche komplexen und fehlerflüchtigen Tätigkeiten an einen Codegenerator gemäß dem modellgetriebenen Paradigma delegiert und somit dem Entwickler abgenommen werden. Aufgrund der steigenden Komplexität des Einsatzes von Sensorknoten und der auf ihnen laufenden Anwendungen bedarf die Testdurchführung einer adäquaten Abstraktion von den technischen Belangen. Dies ist durch die Verwendung ausführbarer visueller Modelle möglich, aus denen einerseits die nötigen Codefragmente zur Testdurchführung generiert werden und durch die andererseits das Verhalten der zu testenden Anwendung visuell dargestellt wird. Durch die mit dem Einsatz ausführbarer formaler Modelle erzielte Abstraktion werden die komplexen Anforderungen beherrschbarer.

Um ScatterUnit um diese Merkmale zu ergänzen, wurde das architekturzentrierte modellgetriebene Rahmenwerk Visual ScatterUnit entwickelt, mit dessen Hilfe das Testszenario als formales Testfallmodell repräsentiert werden kann, ohne dass die verteilte Ausführung der auf den Sensorknoten laufenden Testskripte manuell koordiniert werden muss. Die Aufgabe der Verteilung der Aktionen eines Testszenarios auf die verschiedenen Sensorknoten sowie die Koordination ihrer Ausführung kann gemäß dem AC-MDSD an den Codegenerator delegiert werden, wodurch eine Automatisierung dieser Aktivität erzielt wird. Dies ist möglich, weil die notwendigen Informationen im formalen Modell festgehalten werden. Des Weiteren werden für den Test relevante Informationen in den formalen Modellen dargestellt. So werden die von

ScatterUnit protokollierten Testergebnisse nach der Testdurchführung in demselben formalen Modelldiagramm, welches die Testdurchführung repräsentiert, visualisiert. Auf diese Weise kann das Verhalten der zu testenden Anwendung dem Benutzer visuell dargestellt werden.

Visual ScatterUnit macht es möglich, durch das kognitive visuelle Vorgehen mithilfe der iterativen Verfeinerung des Testfallmodells die Ursache für das Versagen der zu testenden Anwendung zu lokalisieren. Indem Visual ScatterUnit die zielgerichtete Verfeinerung der Testdiagramme ermöglicht, um die vom Softwaretester aufgestellten Hypothesen zur Fehlerlokalisierung zu verifizieren, wird diese Vorgehensweise vereinfacht. Dabei liegt der große Nutzen des modellgetriebenen Paradigmas von Visual ScatterUnit in der Automatisierung des in hohem Maße iterativen Verfeinerungsprozesses des Modells: Werden bei weiterer Verfeinerung neue Entitäten ins Testfalldiagramm eingebettet, muss die Ordnung aller im Diagramm vorhandenen Aktionen entsprechend umgestaltet werden. Dies leistet der entsprechende Code zur Koordination der erneuten Ausführung der Testskripte auf den Sensorknoten. Da dieser Code mit Hilfe von Visual ScatterUnit automatisiert generiert wird, muss der Softwaretester die technische Umsetzung der Modellverfeinerung nicht selbst bewältigen.

Die erzielten Vorteile machen sich insbesondere bemerkbar, wenn man die Herausforderung betrachtet, suspekte Codesegmente bis zur Lokalisierung der Fehlerursache iterativ zu isolieren: Bei der iterativen Verfeinerung des Modells für die erneute Testdurchführung müsste der Benutzer jedes Mal den dazu nötigen Infrastrukturcode neu schreiben. Da dies jedoch mit Hilfe des Codegenerators automatisiert vonstatten geht, reduziert sich der Zeitaufwand für den Implementierungsprozess des Testfalls signifikant. Insgesamt wurde es mit Hilfe von Visual ScatterUnit möglich, verschiedene, für die Implementierung des Testfalls essentielle Aktivitäten zu automatisieren und sie somit dem Benutzer zu ersparen. Dies ist von großem Nutzen, da gerade solche Aktivitäten komplex und zeitaufwändig sind. Statt diese Aufgaben selbst und von Hand zu bewältigen, kann der Tester der WSN-Anwendung seinen Fokus auf andere, für den Test entscheidende Belange legen. Bei der Implementierung des Testfalls ist dies der Entwurf des Testszenarios und bei der Fehlerisolierung die adäquate Verfeinerung des Testfallmodels, um die diesbezüglich aufgestellten Hypothesen zu verifizieren. So wird der angestrebte Nutzen des modellgetriebenen architekturzentrierten Ansatzes beim Testen von Anwendungen drahtloser Sensornetze optimal erreicht.

Der automatisierte Test und die Codierung der Anwendungen müssen gemeinsam nah zum jeweiligen Einsatzszenario des Sensornetzes durchgeführt werden. Die Brücke zur Betriebsnähe liegt in der Integration von Verwaltungsfunktionalitäten in den automatisierten Softwareentwicklungsprozess, so dass dieser nicht isoliert betrachtet wird. Durch die integrierte Verwaltung des Sensornetzes kann während der Anwendungstests und der Erstellung der Sensoranwendungen der Status der einzelnen Knoten visualisiert, konfiguriert, überprüft und überwacht werden. Da die Sensorknoten häufig an Orten platziert werden, zu denen Menschen schwer Zugang haben, muss die Verwaltung sowie die Ausbringung der Softwareartefakte entfernt betrieben werden. Hier ist ein internetbasierter Zugriff von Nöten. Um solche Verwaltungsfunktionalitäten zu realisieren, wurden die Rahmenwerke ScatterPlug und ScatterEditor entwickelt.

Durch das Plugin-orientierte ScatterEditor werden wichtige Verwaltungsfunktionalitäten zur Verfügung gestellt. Beispielsweise können neben der Konfiguration und der Überwachung der Sensorknoten Aktualisierungen der auf den Knoten laufenden Softwaremodule per Funk vorgenommen werden. Die Funktionalitäten können sowohl im lokalen als auch entfernt im internetbasierten Modus in Anspruch genommen werden. So kann der Benutzer mit Hilfe von ScatterFactory und Visual ScatterUnit die Codeartefakte einer Anwendung und deren

Testfälle automatisiert erstellen und anschließend mit Hilfe von ScatterEditor die generierten Codeartefakte internetbasiert auf ausgewählten Sensorknoten im drahtlosen Netz per Funk ausbringen. Ferner können parallel zur automatisierten Erstellung und zum Test einer Anwendung eines bestimmten Sensorknotens die Hardware bzw. Firmwareflaggen (z.B. die Sendestärke) mit Hilfe von ScatterEditor passend zum jeweiligen Testfall konfiguriert werden. Der Fokus von ScatterEditor liegt neben der Verbindung zwischen IP und ScatterWeb auf der Schnittstelle zwischen Mensch und Maschine bei drahtlosen Sensornetzen. ScatterEditor bietet die Möglichkeit, verschiedene Charakteristika und Eigenschaften der WSNs illustrativ und interaktiv zu testen und zu untersuchen.

Neben den Basisdiensten, wie Programmierung und Konfiguration von Sensorknoten, bietet ScatterPlug auch komplexere Dienste an, wie die Emulation und Modellierung von ausgebrachten drahtlosen Sensornetzen im objektorientierten Paradigma, die statistische Analyse der WSN-Umgebung, ebenso wie die Visualisierung der analysierten Daten aus dem WSN und des Status der einzelnen Sensoren. Die statistische Analyse der von einem Sensor erfassten und bearbeiteten Daten in ScatterPlug kann sehr hilfreich sein, um Rückschlüsse auf die Messungsgenauigkeit der Sensorik der einzelnen Sensorknoten zu ziehen. Der Benutzer kann mittels Visual ScatterUnit visuell und automatisiert die Versagensursache in der Anwendung lokalisieren und parallel mit Hilfe der statistischen Analyse gesammelter Sensordaten in ScatterPlug überprüfen, ob das Versagen etwa doch auf die Hardwaresensorik des Knotens zurückzuführen ist und nicht auf die Implementierung der darauf laufenden Anwendung. Ein weiteres wichtiges Merkmal von ScatterPlug ist die Fähigkeit, die gesamte Konfiguration des erstellten drahtlosen Netzes im XML-Format zu speichern. Somit können die verschiedenen Plugins die XML-Daten untereinander importieren bzw. exportieren.

Wie man sieht, arbeiten die Verwaltungsfunktionalitäten von ScatterPlug und ScatterFactory Hand in Hand mit den Funktionalitäten des modellgetriebenen Prozesses und ergänzen daher den Prozess der Softwareentwicklung. Diese Zusammenarbeit, welche ein wichtiges Kennzeichen der Plattform darstellt, wurde architektonisch durch den Plugin-orientierten Aufbau der ScatterClipse-Werkzeugkette gemäß der Eclipse-Plattform realisiert. Folglich weist die Architektur der Plattform einen hohen Grad an Offenheit auf, was ihre Interoperabilität deutlich steigert. Diese Eigenschaft ist essentiell, insbesondere da in der dynamischen WSN-Domäne immer neue Fragestellungen auftauchen.

Der Benutzer kann sich einerseits bestimmter Plugins (Funktionalitäten) unabhängig von den anderen bedienen, wodurch ein *separation of concerns* erreicht wird, andererseits kann er zwischen den verschiedenen Plugins navigieren, was die Kohärenz erhöht. Nichts desto trotz ist für die interaktive Navigation des Benutzers zwischen den vielseitigen sich ergänzenden Komponenten der Werkzeugkette ein Gefüge notwendig, innerhalb dessen der Benutzer die jeweils benötigten Komponenten zielgerichtet und mit besserer Orientierung hinsichtlich seines Gesamtvorhabens bedienen kann. Daher sollten der Zugriff und die Nutzung der unterschiedlichen Plugins bzw. Komponenten der Werkzeugkette zentral geschehen und einer einheitlichen Vorrichtung unterliegen. Zu diesem Zweck wurde die *ScatterClipse Assembly Line* entwickelt, die, wie der Name vermuten lässt, als ein integriertes View-Plugin realisiert wurde, welches die betätigten Aktivitäten der Werkzeugkette in chronologisch strukturierte Abschnitte aufteilt. Jeder Abschnitt besteht aus verschiedenen Aktivitäten, wobei jede Aktivität durch ihr spezielles Plugin realisiert wird. Somit findet eine Visualisierung der Zusammenarbeit der verschiedenen Komponenten der Werkzeugkette während ihrer Inanspruchnahme statt, wodurch dem Benutzer stets die Gesamtübersicht erhalten bleibt. Dies verdeutlicht Abbildung 83, die einen Bildschirmabzug der ScatterClipse Assembly Line zeigt:

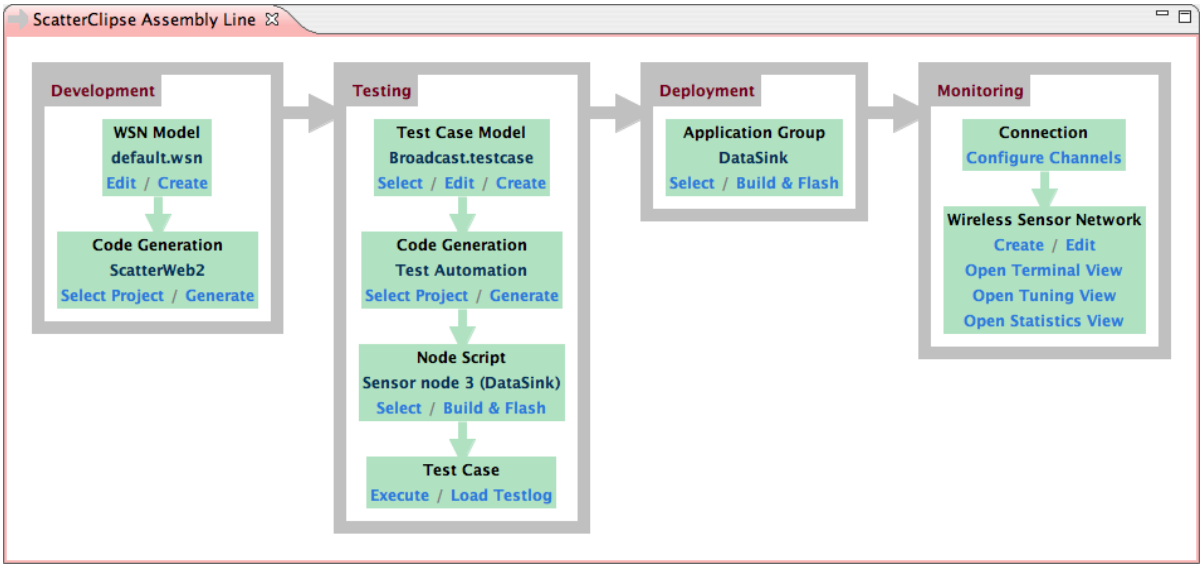


Abbildung 82 Die ScatterClipse Assembly Line [Kame08]

Insgesamt repräsentiert die ScatterClipse Assembly Line die Mächtigkeit der Werkzeugkette, gibt dem Benutzer einen Überblick über die zur Verfügung stehenden Funktionalitäten und erleichtert den Zugriff auf sie. Die einfache Zugriffsmöglichkeit motiviert darüber hinaus dazu, alle Werkzeuge von ScatterClipse zusammen zu benutzen.

Alles in allem wurde mit ScatterClipse eine neuartige Anwendung des modellgetriebenen Paradigmas auf die anspruchsvolle Domäne der drahtlosen Sensornetze erfolgreich demonstriert. Einerseits demonstriert diese Anwendung das Potential des modellgetriebenen architekturzentrierten Ansatzes und anderseits wurde gezeigt, welch hoher Bedarf besteht, gegenwärtig neu entwickelte Ansätze in solch sensitiven Domänen wie drahtlosen Sensornetzen anzuwenden.[19]

Der zweigleisige Einsatz der Modelle – bestehend aus der Kombination der Visualisierung des Anwendungsverhaltens einerseits und dem Zweck der Testautomatisierung der Sensoranwendungen andererseits – stellt die Innovation des Rahmenwerkes dar, welche im Bereich der Softwaremodellierung positiv aufgenommen wurde: „[…] the idea of incorporating the test results in the test case model […] is certainly an interesting contribution that could be of general interest".[20] Meiner Ansicht nach handelt es sich hierbei um einen generalisierbaren Ansatz, der über ein großes Potential verfügt, in unterschiedlichsten Domänen eingesetzt zu werden.

[19] Ein (anonymer) Reviewer des Visual ScatterUnit Papers in ACM/IEEE Models´08: „Paper is interesting subject and interesting application of model driven approach to testing WSNs."
[20] Ein (anonymer) Reviewer des Visual ScatterUnit Papers in ACM/IEEE Models´08.

11. Anhang

11.1. Abkürzungsverzeichnis:

Abkürzung	Begriff
AC-MDSD	Architecture-Centric Model-Driven Software Development
DSDV	Destination-Sequenced Distance-Vector Routing
DSL	Domain Specific Language
EMF	Eclipse Modeling
GMF	Graphical Modeling
M2M	Model to Model Transformation
MDA	Model-Driven Architecture
MDSD	Model-Driven Software Development
MOF	Meta Object Facility
oAW	openArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
RMI	Remote Method Invocation
TTCN	Testing and Test Control Notation
UML	Unified Modeling Language
WSN	Wireless Sensor Network
XMI	XML Metadata Interchange

11.2. Glossar

Domain-Specific Language, Domänensprache (DSL): DSL ist eine Modellierungssprache oder Spezifikationssprache, die für eine Domäne mit einer spezifischen Problemstellung bestimmt ist; eine bestimmte Technik der Repräsentation eines Problems und/oder eine spezielle Lösungstechnik. Der Ausdruck wurde jedoch gängig aufgrund des Aufstiegs von domänenspezifischer Modellierung. [WebDSL]

Meta Object Facility (MOF): MOF ist die Grundlage der Industriestandard-Umgebung von OMG (→ OMG). Modelle können darin von einer Anwendung in eine andere importiert, in ein Netzwerk transportiert und in einem Repository abgelegt und von dort wieder abgefragt werden, wobei sie in verschiedene Formate (darunter XMI, → XMI) serialisiert werden können. Diese Funktionen stehen nicht nur für strukturierte Modelle oder nur für UML-Modelle zur Verfügung, sondern auch für Verhaltensmodelle und Datenmodelle; ebenso für nicht-UML Modellierungssprachen die MOF basiert sind. [WebMOF]

Model-Driven Architecture, Modellgetriebene Architektur (MDA): MDA stellt einen offenen, anbieterunabhängigen Ansatz zur Verfügung. Geschäfts- und Anwendungslogik werden von der unterliegenden Plattformtechnologie getrennt. Plattformunabhängige Modelle einer Anwendung oder Geschäftsfunktionalitäten bzw. -verhalten integrierter Systeme, die mit UML (→ UML) und anderen OMG-Modellierungsstandards (→ OMG) angefertigt wurden, können auf nahezu jeder Plattform realisiert werden. Die Modelle werden in einer Sequenz von Modelltransformationen in ausführbaren Code transformiert. [WebMDA]

Model-Driven Software Development, Modellgetriebene Softwareentwicklung (MDSD): MDSD ist ein gegenwärtig von Entwicklern und Forschern stark beachtetes Entwicklungsparadigma. MDSD bezeichnet die Verwendung von Domänensprachen (→ DSL), um Modelle zu erstellen, welche die Struktur oder das Verhalten einer Anwendung in einer effizienten und problemorientierten Art und Weise ausdrücken. [Stah06]

Object Constraint Language (OCL): OCL ist eine formale Sprache, die zum Ausdruck von UML-Modellen (→ UML) eingesetzt wird. Diese Ausdrücke spezifizieren typischerweise unveränderliche Bedingungen, die für das modellierte System oder für Anfragen über an das Modell gebundene Entitäten eingehalten werden müssen. Wenn OCL-Ausdrücke bewertet werden, haben sie keine Nebeneffekte, d.h. ihre Bewertung kann den Status des dazugehörigen ausführenden Systems nicht verändern. OCL- Ausdrücke werden oft in UML-Modellen verwendet, um semantische und syntaktische Randbedingungen zu spezifizieren, was als Standardtechnik von UML-Profiles (→ UML-Profiles) verwendet wird. [WebOCL]

Object Management Group (OMG): OMG ist ein 1989 gegründetes, gemeinnütziges Konsortium für IT-Standards mit offener Mitgliedschaft. Sie unterstützt IT-Spezifikationen für allgemein kompatible, übertragbare und

wiederverwendbare Unternehmensanwendungen in verteilten und heterogenen Umgebungen. Mitglieder sind sowohl Verkäufer von Informationstechnologie, Endbenutzer, Regierungsbehörden als auch Hochschulen. [WebOMG]

Remote Method Invocation (RMI): RMI bietet ein einfaches und direktes Modell für verteilte Anwendungen mit Hilfe von Java-Objektinstanzen. Java unterstützt das „Einmal-Schreiben-Überall-Verwenden"-Modell. RMI erweitert das Java-Modell, damit es entfernt in Anspruch genommen werden kann. Da von den technischen Details der Verteilung abstrahiert wird, versetzt Java-RMI den Entwickler in die Lage, auf einfache Art verteilte Java-basierte Klient-Server-Anwendungen zu erstellen. [WebRMI]

Software-Entwicklungsprozess: Ein spezielles Vorgehensmodell, das der Steuerung der Softwareentwicklung von der Konzeption bis zum Einsatz im Echtbetrieb (inklusive der im Echtbetrieb anfallenden Anpassungen einer Software) dient. Beispiele sind Wasserfallmodell, Spiralmodell, Rational Unified Process (RUP), modellgetriebene Softwareentwicklung (→ MDSD), etc. [WebEntwicklungsprozess]

Testfall: In der Softwareentwicklung ist ein Testfall ein Set von Bedingungen oder Variablen unter denen ein Tester bestimmt, ob eine Anwendung oder ein Softwaresystem korrekt arbeitet oder nicht. Der Testfall realisiert das vom Softwaretester entworfene Testszenario. Es kann eine Vielzahl von Testfällen erfordern, um festzustellen, ob ein Programm oder System korrekt funktioniert. Testfälle werden oft als Testskripts bezeichnet, insbesondere, wenn sie codiert werden. Codierte Testfälle werden üblicherweise als Test-Suite gesammelt. [WebTestfall]

UML-Profiles: Der Profile-Mechanismus wurde mit dem Zweck definiert, einen leichtgewichtigen Erweiterungsmechanismus für den UML-Standard zur Verfügung zu stellen. In UML 1.1 wurden Stereotype und tagged values (Attribute) als Erweiterungen verwendet, die an die UML-Modelle flexibel angebunden werden können. In nachfolgenden Verbesserungen von UML wurde das Konzept des Profils definiert, um mehr Struktur und Präzision für die Definierung von Stereotypen und tagged values zu erreichen. Die UML 2.0 Spezifikation hat die Technik der Profiles weitergetrieben, indem diese als eine spezifische Metamodellierungstechnik definiert wurde. [WebProfiles]

Unified Modeling Language (UML): UML ist die meist verwendete Spezifikation der OMG (→ OMG). Struktur, Verhalten und Architektur von Anwendungen lassen sich damit ebenso modellieren wie Geschäftsprozesse und Datenstrukturen. UML stellt die Grundlage für die Modellgetriebene Architektur (→ MDA) dar. UML hat sich als Standardtechnik für Objektorientierte Programmierung durchgesetzt. [WebUML]

XML Metadata Interchange (XMI): XMI ist ein XML-basiertes Austauschformat, durch das sich (Meta-)Modelle verschiedener Technologien serialisieren lassen. XMI bietet Regeln, durch die ein Schema für jedes gültige, XMI-übertragbare, MOF-basierte Metamodell generiert werden kann. [WebXML]

11.3. Benutzerinteraktion

Im Folgenden wird die Benutzerinteraktion mit ausgewählten Komponenten der ScatterClipse-Werkzeugkette demonstriert, um den Aspekt *Human-Computer Interfaces* für WSN in der Werkzeugkette zu illustrieren.

11.3.1. Der Wizard zur Einrichtung des WSN

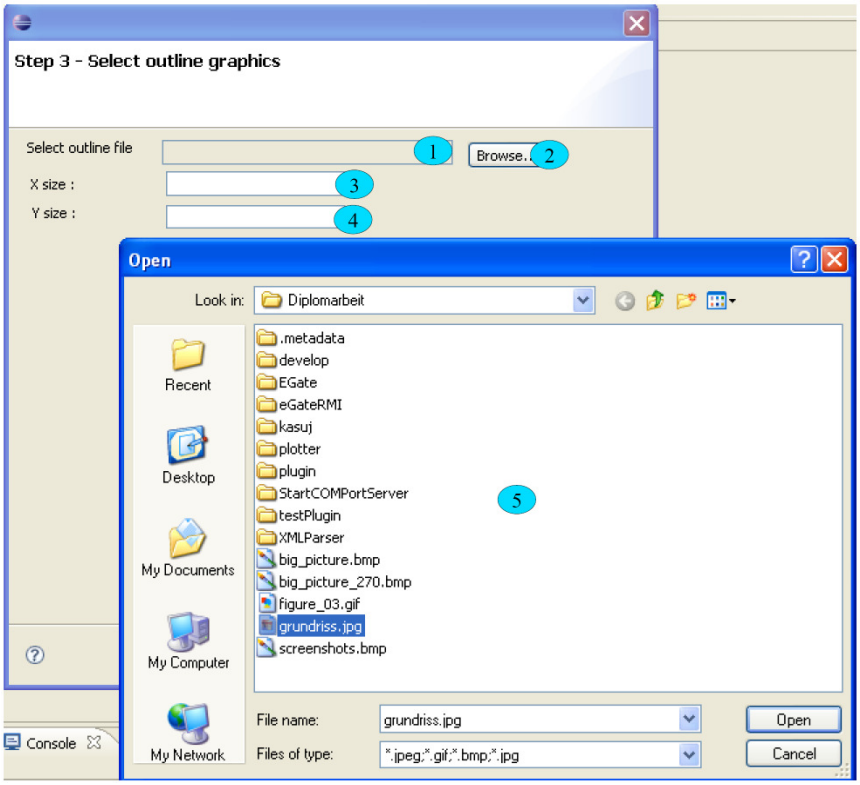


Abbildung 83 Der Wizard von ScatterPlug zur Einrichtung des WSN [Lesz06]

Im ersten Schritt (Abbildung 84 oben links) wird das Projekt des zu errichtenden WSNs angelegt. Im folgenden Schritt (oben rechts) wird die Konfiguration der Datenbank vorgenommen, und im letzten Schritt (Mitte) kann abschließend die Grafik des WSN-Grundrisses und die dazu nötige Skalierung gemäß der realen Entfernungen angegeben werden.

11.3.2. Das ScatterFlash-Plugin: Kompilierung und Initialisierung der Software (Deployment)

Im ersten Teil der Abbildung 85 von ScatterFlash wird der *package explorer* gezeigt, in dem die aus dem automatisierten Codegenerierungsprozess resultierenden Dateien abgelegt werden (1). In (2) sieht man den Editor des Plugins, in dem der Code der Sensorknoten bearbeitet werden kann. Während der Bearbeitung werden verschiedene Dienste durch das ScatterFlash-Plugin zur Hilfestellung angeboten. So ist es möglich, den Quellcode durch Hervorheben von Syntaxkomponenten zu editieren. Fehlermeldung werden während des Kompilierungsprozesses im Konsolefenster von Eclipse angezeigt (4). Des Weiteren wurde das ANT-Plugin in ScatterFlash als Standardkomponente integriert (3), um eine bessere Handhabung der Kompilierung und Ausbringung des Codes in die Sensorknoten über die serielle Schnittstelle zu erreichen.

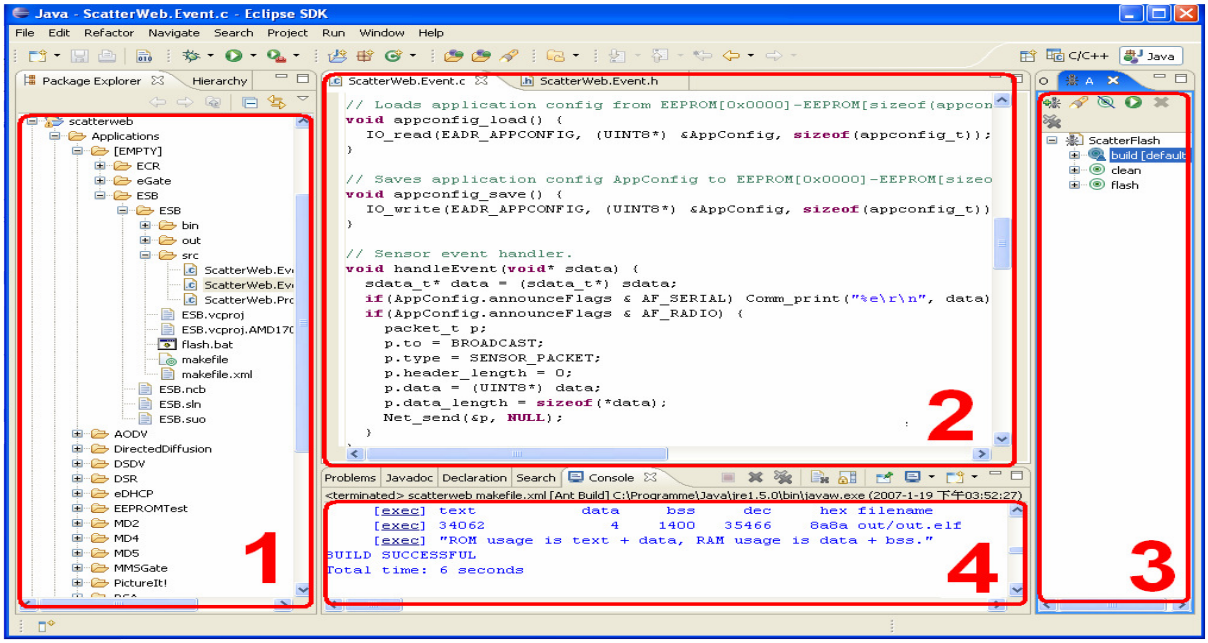


Abbildung 84 Das ScatterFlash-Plugin zur Kompilierung und Initialisierung der generierten Softwareartefakte [Ding07]

Zu diesem Zweck unterstützt das Ant-Plugin sogenannte *Targets*, die als XML-Elemente vorliegen. In solchen Targets können beliebige Aktionen bzw. Steuerbefehle angegeben werden, die externe Dienste bzw. Funktionalitäten aufrufen. Hierzu werden mittels der Targets diejenigen Aktionen der Makfiles angegeben, die die entsprechenden Funktionalitäten (zur Kompilierung und zum seriellen Flashing) der mspgcc-Software aufrufen. Dabei handelt es sich um drei Targets (siehe Abbildung 85 oben rechts).

1. *build* zur Kompilierung des C-Quellcodes eines ausgewählten Ordners
2. *flash* zur Ausbringung von kompilierten Codeartefakten (Hex-Files) auf den Sensorknoten über die serielle Schnittstelle
3. *clean* zur Löschung aller bereits kompilierten Artefakte

11.3.3. OTA-Flashing

Abbildung 86 zeigt einen Bildschirmabzug der Benutzerschnittstelle des OTA-Flashings:

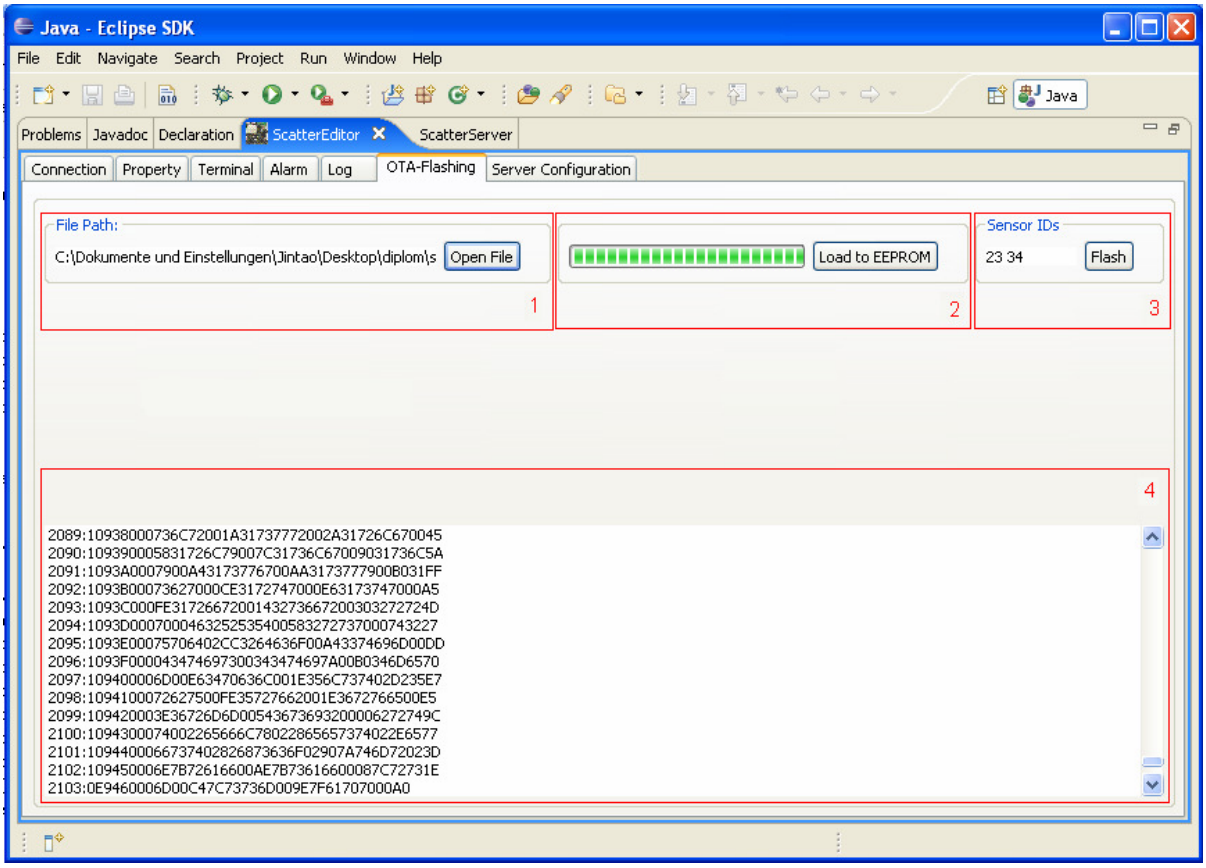


Abbildung 85 Der OTA-Flashing-Reiter [Ding07]

Wie in Abbildung 61 gezeigt wurde, werden bei der Betätigung der Refresh-Funktion sämtliche Sensorknoten-IDs in der Tabelle aufgelistet, die vom eGate aus erreichbar sind. Nun kann der Benutzer im Reiter des OTA-Flashings die IDs derjenigen Sensorknoten eingeben, zu denen die Softwareartefakte ausgebracht werden sollen (3). Die Auswahl der Softwareartefakte findet mit Hilfe einer Auswahl des Verzeichnisbaums *file chooser* (1) statt, die es dem Benutzer ermöglicht, flexibel und überschaubar zwischen den unterschiedlichen generierten Softwareartefakten zu navigieren, da diese, wie in 5.4 erläutert, nach dem Generierungsprozess in spezifische Verzeichnisse geschrieben werden, die vom Benutzer innerhalb der Templates festgelegt werden. Nach der Auswahl des als Hex-File vorliegenden Softwareartefakts aus dem entsprechenden Verzeichnis wird dieses zuerst in den EEPROM des eGates geladen. Der Benutzer kann das Voranschreiten des Ladevorgangs mittels der progress bar verfolgen (2). Währenddessen werden in diesem Zusammenhang relevante Nachrichten in (4) ausgegeben. Abschließend kann das Softwaremodul (Inhalt des EEPROMs des eGates) per Funk zu den in (3) ausgewählten Sensorknoten gesendet werden. Der daraus entstehende Nachrichtenaustausch wird in (4) ausgegeben. Soll das Softwaremodul ins eGate selbst installiert werden, so muss in (3) statt der Eingabe der Sensor-IDs „eGate“ eingetippt werden

11.3.4. Der Alarm-Reiter

Abbildung 87 zeigt einen Bildschirmabzug des Alarm-Reiters:

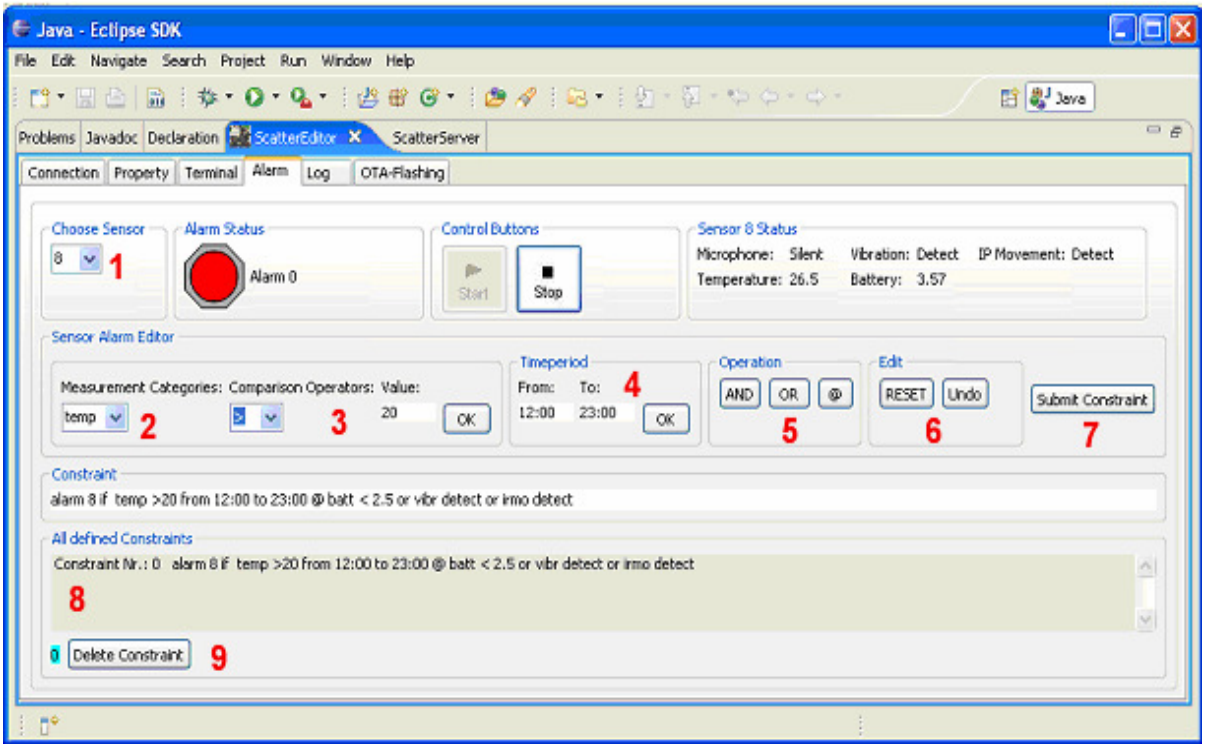


Abbildung 86 Der Alarm-Reiter [Ding07]

Im Alarm-Reiter kann der Benutzer das Alarmsystem konfigurieren und wird illustrativ über erfüllte Bedingungen im WSN informiert. Die Interaktion des Benutzers mit dem Reiter ist ein weiteres Beispiel für die Implementierung der Schnittstelle zwischen Mensch und Maschine in ScatterWeb. Zuerst wählt der Benutzer die IDs derjenigen Sensorknoten aus, die von der Alarmbedingung erfasst werden sollen (siehe (1) im obigen Bildschirmabzug). Zunächst werden die zu beobachtenden Messgrößen im Alarmeditor (2) selektiert. Die ausgewählten Messgrößen werden dann gemäß der vorgestellten Grammatik der Alarmsprache mittels Vergleichsoperatoren verknüpft (3), wodurch sich Alarmprädikate bilden. Dann werden den ausgewählten Messgrößen eindeutige Werte zugewiesen (3). Zuletzt wird die Zeitspanne definiert, innerhalb derer das Alarmprädikat erfüllt werden muss (4). Jedes formulierte Alarmprädikat kann mit den Operatoren OR und AND an ein weiteres geknüpft werden (5).

11.3.5. Benutzerinteraktion im entfernten Modus

Ein weiterer Aspekt zur Steigerung der Sicherheitsbelange beim Einsatz des entfernten Modus ist die Funktionalität der serverseitigen Firewall. Dies zeigt sich deutlich anhand der IP-Liste (2) in der Abbildung 88, die einen Bildschirmabzug der serverseitigen Konfigurationsfunktionalität für die Klientenzugriffsverwaltung zeigt.

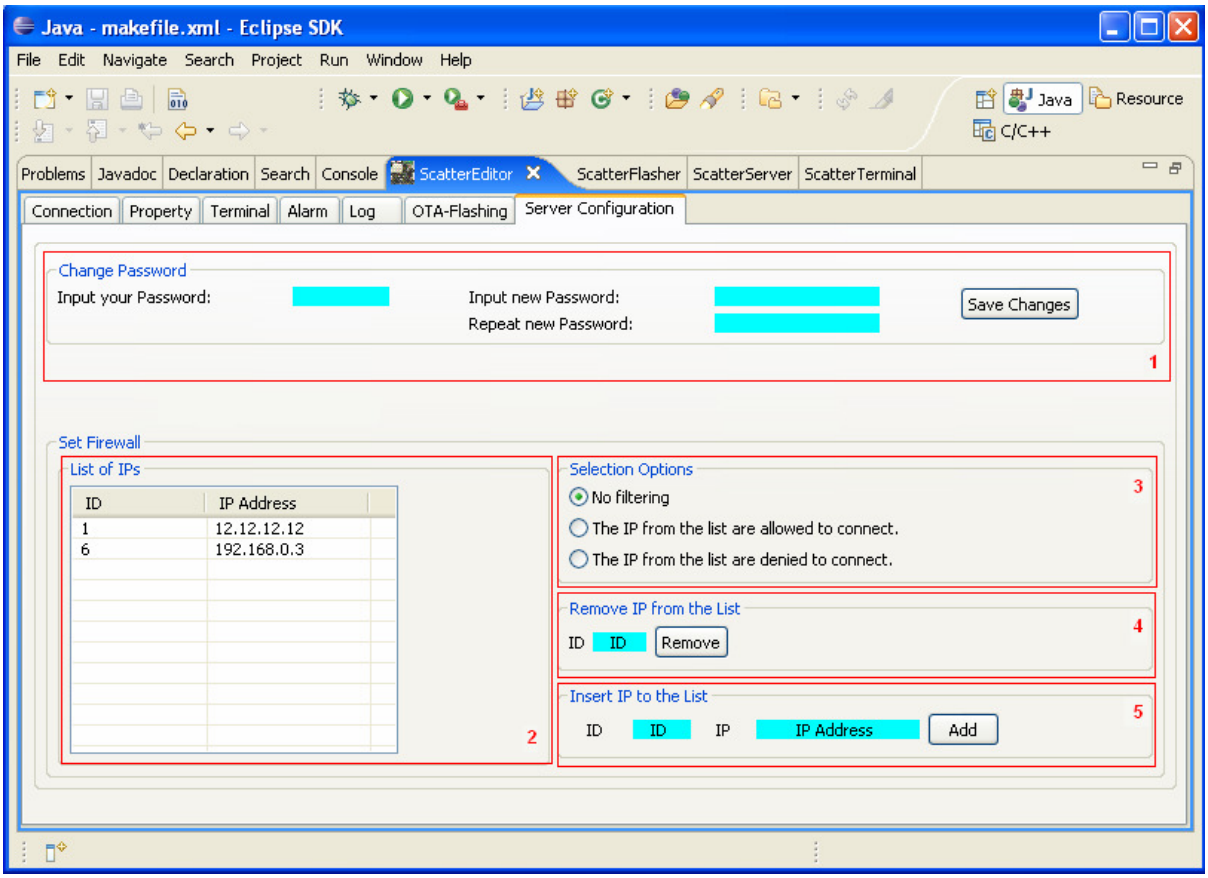


Abbildung 87 Serverkonfiguration zur Klientenverwaltung [Ding07]

In (1) wird dem Administrator die Möglichkeit gegeben, das automatisch generierte Passwort zu ändern. In (3) kann die Firewall eingestellt werden, indem festgelegt wird, nach welchem Modus die eingehenden IPs der anfragenden Klienten gefiltert werden. So ist es möglich, die in der Liste eingetragenen IPs auszuschließen, so dass alle von ihnen stammenden Nachrichten zurückgehalten werden. Alternativ können nur diejenigen Anfragen zugelassen und somit bearbeitet werden, die aus Rechnern stammen, deren IPs in der Liste (2) eingetragen sind. Die Eingabe und Entfernung der IPs aus der Liste geschieht der Reihe nach mittels der Funktionen (4) und (5).

11.4. Das Metamodell der Domänensprache

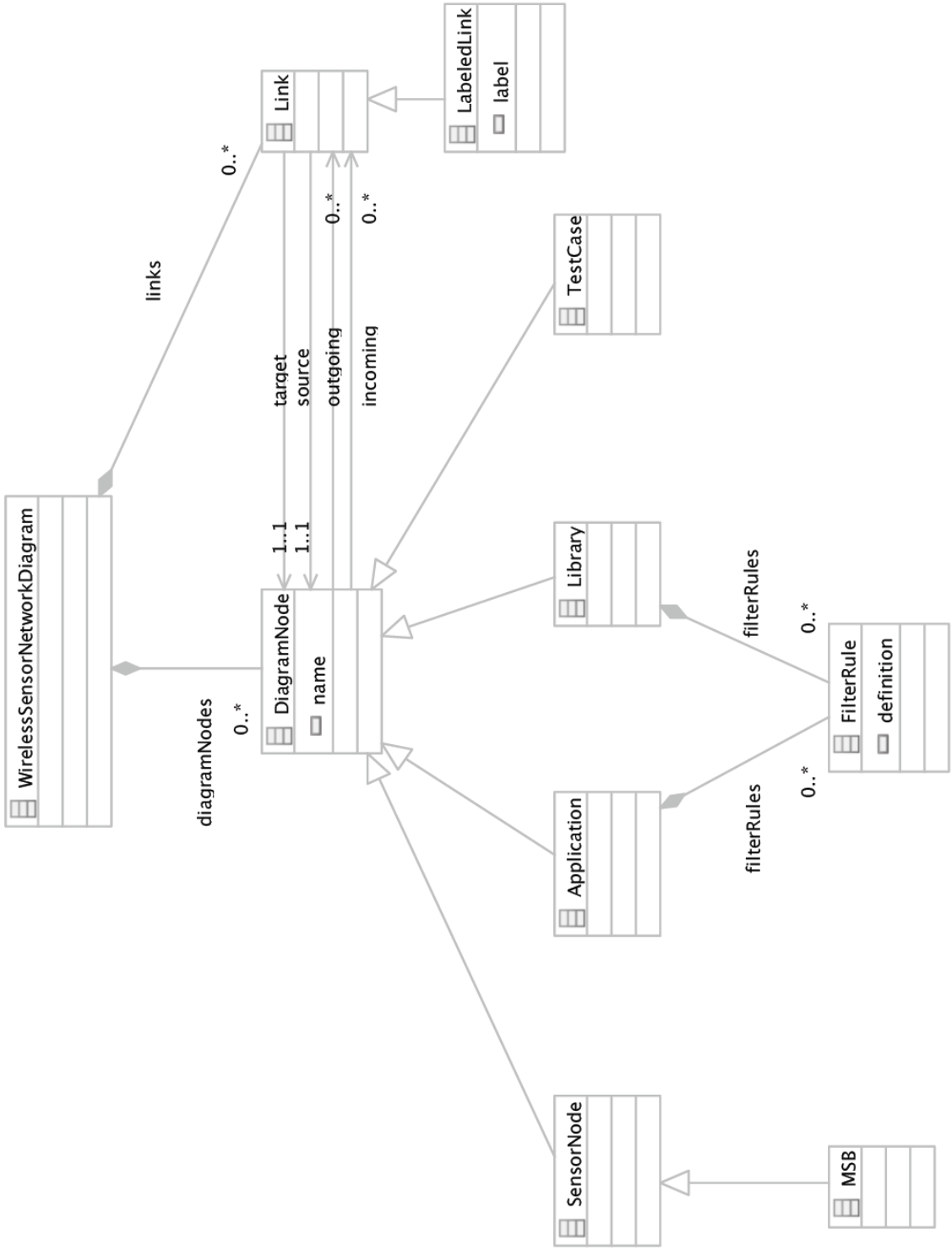


Abbildung 88 Das Metamodell der DSL in ScatterFactory2 [Kame08]

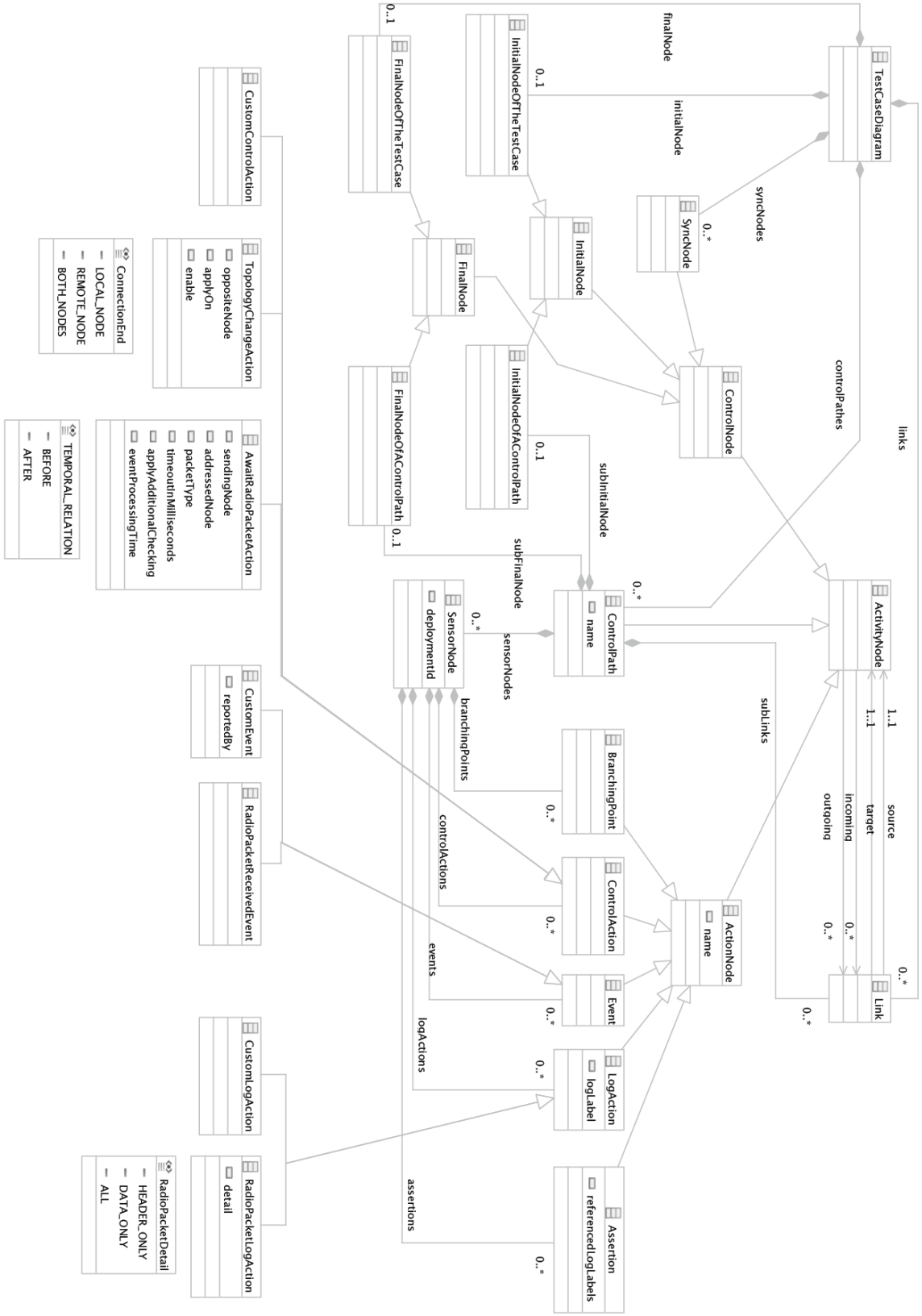


Abbildung 89 Das Metamodell der DSL zum automatisierten Anwendungstest in Visual ScatterUnit [Kame08]

11.5. Die in Checks formulierten Randbedingungen

11.5.1. Randbedingungen in Visual ScatterUnit

Im Folgenden wird eine breite Auswahl der in Visual ScatterUnit definierten Checks-Ausdrücke aufgelistet [Kame08]. Die in den Ausdrücken vorkommenden Modellvariablen beziehen sich auf die Entitäten des für Visual ScatterUnit definierten Metamodells, welches in der Abbildung 90 dargestellt wurde.

context testcase::TestCaseDiagram

ERROR ”Exactly one initial node must exist.” :this.initialNode != null;

context testcase::TestCaseDiagram

ERROR ”Exactly one final node must exist.” :this.finalNode != null;

context testcase::TestCaseDiagram

ERROR ”At least one control path must exist.” :this.controlPathes.size > 0;

context testcase::TestCaseDiagram

ERROR ”Circular links are not allowed.” :this.initialNode != null ?

this.initialNode.checkForCircularLinks() == false : true;

context testcase::InitialNodeOfTheTestCase

ERROR ”The initial node must not have any incoming links.” :

this.incoming.size == 0;

context testcase::InitialNodeOfTheTestCase

ERROR ”The initial node must have exactly one outgoing link.” :

this.outgoing.size == 1;

context testcase::FinalNodeOfTheTestCase

ERROR ”The final node must have exactly one incoming link.” :

this.incoming.size == 1;

context testcase::FinalNodeOfTheTestCase

ERROR ”The final node must not have any outgoing links.” :

this.outgoing.size == 0;

context testcase::Link ERROR ”The source node of this link is not specified.” :this.source != null;

context testcase::Link

ERROR ”The target node of this link is not specified.” :

this.target != null;

context testcase::ControlPath

ERROR ”A control path must have one incoming link.” :this.incoming.size == 1;

context testcase::ControlPath

ERROR ”A control path must have at maximum one outgoing link.” :

this.outgoing.size <= 1;

context testcase::ControlPath **ERROR** ”A control path must have a name.” :

this.name.length > 0;

context testcase::ControlPath **ERROR** ”The name of a control path must be unique.” :

this.diagram.controlPathes.select(e | e != this).name.contains(this.name) == false;

context testcase::ControlPath

ERROR ”The name must only contain alphabetic and numeric letters.” :

this.name.matches(”[\\p{Alpha}\\p{Digit}]+”);

context testcase::ControlPath

ERROR ”A control path and its branching points must be linked to distinct nodes.” :

this.allOutgoing().target.size == this.allOutgoing().target.toSet().size;

context testcase::ControlPath

WARNING ”The control path has no successor.” :this.allOutgoing().size > 0;

context testcase::TopologyChangeAction

ERROR ”The id of the opposite node must be greater than zero.” :this.oppositeNode > 0;

context testcase::TopologyChangeAction

ERROR ”The id of the opposite node must be less than 255.” :this.oppositeNode < 255;

context testcase::TopologyChangeAction

ERROR ”The id of the opposite node must be different from the local node id.” :this.oppositeNode != this.sensorNode.deploymentId;

context testcase::RadioPacketReceivedEvent

ERROR ”A preceding await radio packet action must be executed on the same sensor node to report the event.” :this.hasPrecedingAwaitRadioPacketAction();

context testcase::RadioPacketLogAction

ERROR ”A preceding radio packet received event must be executed on the same sensor node.” :this.hasPrecedingRadioPacketReceivedEvent();

context testcase::BranchingPoint

ERROR ”A branching point must not have any incoming links.” :

this.incoming.source.forAll(e | e.metaType != testcase::InitialNodeOfTheTestCase && e.metaType != testcase::SyncNode && e.metaType != testcase::ControlPath);

context testcase::BranchingPoint

ERROR ”A branching point must have at maximum one outgoing link.” :

this.viewableOutgoingSuper().size <= 1;

context testcase::BranchingPoint

ERROR ”A branching point must not be linked to the final node.” :

this.outgoing.size > 0 ?

this.outgoing.target.forAll(e | e.metaType != testcase::FinalNodeOfTheTestCase) :true;

context testcase::BranchingPoint

ERROR ”A branching point must not be linked to the final node.” :

this.outgoing.size > 0 ?

this.outgoing.target.select(e | e.metaType == testcase::SyncNode).outgoing

.notExists(e | e.target.metaType == testcase::FinalNodeOfTheTestCase) :true;

context testcase::BranchingPoint

ERROR ”A branching point must only be linked to a control path which is executed

on a different sensor node.” :

this.branch() != null && this.branch().target.metaType == testcase::ControlPath ?

((testcase::ControlPath)this.branch().target).getDeploymentIdForFirstAction()

!= this.deploymentId() :true;

context testcase::BranchingPoint

ERROR ”A branching point must only be linked to control pathes which are executed on different sensor nodes.” :this.branch() != null && this.branch().target.metaType == testcase::SyncNode ?

((List[testcase::ControlPath])((testcase::SyncNode)this.branch().target).outgoing.target

.select(e | e.metaType == testcase::ControlPath)).getDeploymentIdForFirstAction()

.notExists(e | e == this.deploymentId()) :true;

context testcase::Assertion

ERROR ”An assertion must at least apply one log label.” : hasReferencedLogLabels(this.referencedLogLabels);

context testcase::Assertion

ERROR ”Log labels referenced by an assertion must be provided by log actions.” : this.validateReferencedLogLabels();

context testcase::SyncNode

ERROR ”A sync node must have at least one incoming link.” :this.incoming.size > 0;

context testcase::SyncNode

ERROR ”A sync node must have at least one outgoing link.” :this.outgoing.size > 0;

context testcase::SyncNode

ERROR ”A sync node must not be linked to another sync node.” :this.outgoing.target.select (e | e.metaType == testcase::SyncNode).size == 0;

context testcase::SyncNode

WARNING ”A sync node should have at least two both incoming or both outgoing links.” :this.incoming.size == 0 || this.outgoing.size == 0 ? true : this.incoming.size + this.outgoing.size != 2;

context testcase::SyncNode

ERROR ”A sync node to which the initial node is linked to must not have other predecessors.” :this.incoming.source.exists(e | e.metaType ==

testcase::InitialNodeOfTheTestCase) ? this.incoming.size == 1 : true;

context testcase::SyncNode

ERROR ”A sync node linked to the final node must not have other successors.” :

this.outgoing.target.exists(e | e.metaType == testcase::FinalNodeOfTheTestCase) ?

this.outgoing.size == 1 : true;

context testcase::SyncNode

ERROR ”The control pathes a sync node is linked to must be executed on distinct sensor nodes.” :((List[testcase::ControlPath])this.outgoing.target.select(e | e.metaType

== testcase::ControlPath)).getDeploymentIdForFirstAction().size ==

((List[testcase::ControlPath])this.outgoing.target.select(e | e.metaType

== testcase::ControlPath)).getDeploymentIdForFirstAction().toSet().size;

context testcase::SyncNode

WARNING ”Some control pathes linked to the sync node are executed on the same sensor node.”

```
:((List[testcase::ControlPath])this.incoming.source.select(e | e.metaType
== testcase::ControlPath)).getDeploymentIdForLastAction().size ==
((List[testcase::ControlPath])this.incoming.source.select(e | e.metaType
== testcase::ControlPath)).getDeploymentIdForLastAction().toSet().size;
context testcase::Link
ERROR ”The source node of this link is not specified.” :
this.source != null;
context testcase::Link
ERROR ”The target node of this link is not specified.” : this.target
```

11.5.2. Randbedingungen in ScatterFactory2

Im Folgenden wird eine breite Auswahl der in ScatterFactory2 definierten Checks-Ausdrücke aufgelistet. [Kame08] Die in den Ausdrücken vorkommenden Modellvariablen beziehen sich auf die Entitäten des für ScatterUnit definierten Metamodells, welches in Abbildung 89 dargestellt wurde.

context WSN::WirelessSensorNetworkDiagram

ERROR ”The diagram must contain at least one sensor node.” :

this.diagramNodes.select(e | e.metaType.superTypes.contains(WSN::SensorNode)) .size > 0;

context WSN::DiagramNode

ERROR ”This node must have a name.” :

this.metaType.superTypes.contains(WSN::SensorNode) ? true : this.name.length > 0;

context WSN::DiagramNode

ERROR ”The name must only contain alphabetic and numeric letters.” :

this.name.matches(”[\\p{Alpha}\\p{Digit}] ”);

context WSN::SensorNode

ERROR ”A sensor node must be linked to exact one application.” :

this.linkedNodes().select(e | e.metaType == WSN::Application).size == 1;

context WSN::SensorNode

ERROR ”The name of a sensor node must be unique.” :

this.linkedNodes().selectFirst(e | e.metaType == WSN::Application) == null ? true :

this.linkedNodes().selectFirst(e | e.metaType == WSN::Application)

.linkedNodes().select(e | e.metaType.superTypes.contains(WSN::SensorNode)

&& e.name == this.name).size == 1;

context WSN::FilterRule

ERROR ”The definition of a filter rule must not be empty.” :

this.definition != null && this.definition.length > 0;

context WSN::TestCase

ERROR ”A test case must be assigned to at least one sensor node.” :

this.linkedNodes().select(e | e.metaType.superTypes.contains(WSN::SensorNode)).size

> 0;

context WSN::TestCase

ERROR ”Some sensor node ids are assigned to more than one sensor node.” :

this.outgoing.size + this.incoming.size == 0 ? true :

((Set[WSN::LabeledLink])this.outgoing.union(this.incoming)).label.hasDuplicateIds() == false;

context WSN::Library

ERROR ”A library must be linked to at least one application.” :

this.linkedNodes().select(e | e.metaType == WSN::Application).size > 0;

context WSN::Application

ERROR ”An application must be linked to at least one sensor node.” :

this.linkedNodes().select(e | e.metaType.superTypes.contains(WSN::SensorNode)).size > 0;

context WSN::Application

ERROR ”The name of an application must be unique.” :

this.name == null || this.name.length == 0 ? true :

this.diagram.diagramNodes.select(e | e.metaType == WSN::Application && e != this)

.name.contains(this.name) == false;

context WSN::Application

WARNING ”An application should be linked to the library \”scatterweb\”.” :

this.linkedNodes()

.select(e | e.metaType == WSN::Library && e.name == ”scatterweb”).size > 0;

context WSN::Link

ERROR ”The source node of this link is not specified.” :this.source != null;

context WSN::Link

ERROR ”The target node of this link is not specified.” :this.target != null;

context WSN::TestCase

ERROR ”A test case must be assigned to at least one sensor node.” :

this.linkedNodes().select(e | e.metaType.superTypes.contains(WSN::SensorNode)).size > 0;

context WSN::TestCase

ERROR ”Some sensor node ids are assigned to more than one sensor node.” :
this.outgoing.size + this.incoming.size == 0 ? true :
((Set[WSN::LabeledLink])this.outgoing.union(this.incoming))
.label.hasDuplicateIds() == false;

11.6. ScatterClipse im Einsatz

Vielfältige Einsatzbeispiele der unterschiedlichen Komponenten der ScatterClipse Werkzeugkette sowie deren Zusammenarbeit finden sich im vierteiligen Screencast unter:
http://page.mi.fu-berlin.de/saad/ScatterClipse/video.htm
Dabei wird jeder einzelne Schritt ausführlich kommentiert und erläutert. Ferner finden sich weitere Beispiele im mehrteiligen Entwickler- bzw. Benutzerhandbuch, welches von der Webseite des Projektes heruntergeladen werden kann. Die Seite des ScatterClipse-Projektes findet sich unter:
http://page.mi.fu-berlin.de/saad/ScatterClipse/scatterclipse.htm

11.7. Verzeichnisse

11.7.1. Abbildungsverzeichnis

1. Abbildung 1 Der MDA-Prozess 7
2. Abbildung 2 Der AC-MDSD-Prozess 9
3. Abbildung 3 Die Eclipse-Plattform als OSGi-Server 10
4. Abbildung 4 Die Elemente von Ecore als Klassenhierarchie 12
5. Abbildung 5 Die generative Schicht von EMF 13
6. Abbildung 6 Modell-Validierung im GMF Editor 15
7. Abbildung 7 Die automatisierte Erstellung des Editors im GMF 16
8. Abbildung 8 Die Architektur und Technologien von oAW 17
9. Abbildung 9 Ein einfaches mittels Checks zu validierendes Modell 18
10. Abbildung 10 Die Architektur der Live-Validierung 19
11. Abbildung 11 Die Template-Sprache Xpand von oAW 21
12. Abbildung 12 Die Workflow-Konfiguration von ScatterFactory 22
13. Abbildung 13 Die in ScatterClipse eingesetzen Werkzeuge und Technologien im Kontext des MDD-Prozesses 24
14. Abbildung 14 Zusammenarbeit zwischen ScatterUnit und Visual ScatterUnit 29
15. Abbildung 15 Aktivitäten beim Einsatz von Visual ScatterUnit 31
16. Abbildung 16 Der generative Editor von Visual ScatterUnit 32
17. Abbildung 17 Modellierung des Testverlaufs mit Hilfe von Aktivitätsdiagrammen 33
18. Abbildung 18 Die beteiligten Komponenten einer typischen ScatterWeb-Anwendung 34
19. Abbildung 19 Änderung der Topologie während der Testdurchführung 37
20. Abbildung 20 Aktivitätsdiagramm zur Modellierung des Verlaufs des Testszenarios 40
21. Abbildung 21 Das verfeinerte Diagramm der Aktivität "Change Topology" aus Abbildung 20 40
22. Abbildung 22 Verfeinertes Diagramm der Aktivität "SendSecondPaket" 41
23. Abbildung 23 Verfeinerungsdiagramm der Aktivität „AwaitSecondPaket“ 42
24. Abbildung 24 Visualisierung der Fehlermeldung im Modell durch den Einbau von entsprechenden Icons Links der Aktionen 45
25. Abbildung 25 Iterative Vorgehensweise bis der Fehler im Code aufgefunden wurde 46
26. Abbildung 26 Der Prozess zur Isolierung von Codefehlern innerhalb der zu testenden Anwendung 47
27. Abbildung 27 Das verfeinerte Diagramm des Testfalls 48
28. Abbildung 28 Der relevante Ausschnitt des Metamodells (links), das die Entitäten des Aktivitätsdiagramms (rechts) sowie deren Zusammenhänge veranschaulicht 51
29. Abbildung 29 Assoziationen zwischen den Diagrammentitäten (Ausschnitt aus dem Metamodell, UML-Notation) 52
30. Abbildung 30 Verbindungen dürfen bei keinem Startknoten enden (links). Verbindungen dürfen ihren Ausgangspunkt nicht in einem Endzustand haben (rechts). 53
31. Abbildung 31 Der relevante Ausschnitt des Metamodells in UML Notation (links) der Entitäten des Detaillierungsdiagramms (rechts) 54
32. Abbildung 32 Die Markierungen (hier Nutzlast) fungieren als Bindungsmecha nismus zwischen den Protokolleinträgen und deren Prüfroutinen. Die Meldungen

der Überprüfungen werden anschließend im Modell angezeigt (rechts). 56
33. Abbildung 33 Ausschnitt des attributierten Metamodells des Detaillierungsdiagramms 57
34. Abbildung 34 Der generative Editor des Rahmenwerks 58
35. Abbildung 35 Live-Validierung deckt den ungültigen Namen einer Aktivität auf 59
36. Abbildung 36 Live-Validierung deckt auf, dass eine Aktivität keinen Vorgänger besitzt 60
37. Abbildung 37 Live-Validierung entdeckt eine unzulässige parallele Ausführung 61
38. Abbildung 38 Die Laufzeitumgebung der Sensorknoten ohne (links) und mit Erweiterung (rechts) 66
39. Abbildung 39 Markierungen zur Visualisierung der Testergebnisse in der Modellentität 67
40. Abbildung 40 Die Architektur von ScatterFactory 71
41. Abbildung 41 Das Metamodell von ScatterFactory 73
42. Abbildung 42 Der ScatterFactory Editor 75
43. Abbildung 43 Modellierungsbeispiel einer Anwendung 76
44. Abbildung 44 Gleichzeitiger Ablauf des Modellierungs- und des Generierungsprozesses innerhalb von Eclipse 79
45. Abbildung 45 Die generierten Artefakte in ScatterFactory 80
46. Abbildung 46 Das Modell einer Laufzeitumgebung zweier MSB Anwendungen 83
47. Abbildung 47 Eine Laufzeitumgebung mit einem assoziierten Testfall 85
48. Abbildung 48 Das mit der Laufzeitumgebung assoziierte Diagramm des Testfalls TestCollectionProcess 86
49. Abbildung 49 Das Datenmodell in ScatterWeb for Java 89
50. Abbildung 50 Nachrichtenaustausch in ScatterWeb for Java 90
51. Abbildung 51 Einsatz von Log4j zur Protokollierung der Nachrichten 91
52. Abbildung 52 Automatisierte Datenserialisierung mit Hilfe des Castor-Rahmenwerks 92
53. Abbildung 53 Eclipse Projekt für das ScatterWeb WSN 93
54. Abbildung 54 XML als Austauschformat in der Plugin-orientierten Architektur 93
55. Abbildung 55 Main View 94
56. Abbildung 56 Visualisierung der Sensorknoten im Main View 95
57. Abbildung 57 Merkmale des Statistik Views 97
58. Abbildung 58 Kommunikation zwischen Statistik View und JfreeChart 98
59. Abbildung 59 Architektur von ScatterEditor 101
60. Abbildung 60 Internetbasierter Zugriff auf die Dienste von ScatterEditor 102
61. Abbildung 61 Connection-Reiter vor (oben) und nach (unten) dem Verbindungsaufbau 104
62. Abbildung 62 Property-Reiter 106
63. Abbildung 63 Der Terminal bietet eine befehlsgestützte Konfiguration der Sensorknoten 107
64. Abbildung 64 Stapeldatenstruktur zur Verwaltung der bereits eingegebenen Befehle 108
65. Abbildung 65 Beispiel eines Alarmausdrucks (oben) und dessen Syntaxbaum (unten) 110
66. Abbildung 66 Drei rekursive Ausdrücke der Alarmsprache 111

67. Abbildung 67 Einsatz von ScatterEditor als Klient 112
68. Abbildung 68 Die Modelldiagramme der Anwendungen Empty (oben) und Actor (unten) 115
69. Abbildung 69 Die Laufzeitumgebung der Sensorarchitektur mit (b) und ohne (a) Testautomatisierung 117
70. Abbildung 70 Erweiterte Laufzeitumgebung des Sensorknotens zur Testautomatisierung 119
71. Abbildung 71 Speicherplatzbedarf der Testautomatisierung 120
72. Abbildung 72 Diagramme des entwickelten Testfalls 123
73. Abbildung 73 Diagramme des entwickelten Testfalls (Fortsetzung) 124
74. Abbildung 74 Protokollierung und Analyse der Routingtabelle zum Zeitpunkt der Versendung des Paketes 125
75. Abbildung 75 Ergänzung des Detaildiagramms ErstesPaketSenden um eine Prüfroutine zur Überprüfung der Next-Hop-Information für Schritt 3 127
76. Abbildung 76 Ergänzung zum Detaildiagramm Init1UndWarten (Warten auf den Empfang der Routing-Information über den Empfänger des Datenpakets für Schritt 4) 128
77. Abbildung 77 Austausch der Routinginformationen über den Zielknoten (Knoten 4) 130
78. Abbildung 78 Das erweiterte Detaildiagramm Init1UndWarten für die Protokollierung des Eintrags der Routingtabelle vor und nach dem Empfang der betroffenen Routinginformation 131
79. Abbildung 79 Benötigte Codegröße (in Codezeilen) für die Testfallentwicklung 133
80. Abbildung 80 Benötigte Codezeilen für die automatisierte Fehlersuche 134
81. Abbildung 81 Anstieg des benötigten Codes (in Codezeilen) zur Fehlersuche vor und nach der Fehlerlokalisierung und der mehrfachen Verfeinerung des Testfallmodells 135
82. Abbildung 82 Die ScatterClipse Assembly Line 155
83. Abbildung 83 Der Wizard von ScatterPlug zur Einrichtung des WSN 161
84. Abbildung 84 Das ScatterFlash-Plugin zur Kompilierung und Initialisierung der generierten Softwareartefakte 162
85. Abbildung 85 Der OTA-Flashing-Reiter 163
86. Abbildung 86 Der Alarm-Reiter 164
87. Abbildung 87 Serverkonfiguration zur Klientenverwaltung 165
88. Abbildung 88 Das Metamodell der DSL in ScatterFactory2 166
89. Abbildung 89 Das Metamodell der DSL zum automatisierten Anwendungstest in Visual ScatterUnit 167

11.7.2. Programmauszugsverzeichnis

1. Programmauszug 1 Ein einfacher Checks-Ausdruck 19
2. Programmauszug 2 Ausdrücke der Sprache Checks zur Überprüfung der semantischen Regel des Vorhandenseins eines Start- und Endknotens im Aktivitätsdiagramm zur Modellierungszeit 52
3. Programmauszug 3 OCL-Ausdruck zum Start- und Endknoten des Aktivitätsdiagramms 52
4. Programmauszug 4 OCL-Ausdruck zum Start- und Endknoten des Aktivitätsdiagramm 53
5. Programmauszug 5 Checks-Ausdruck zur syntaktischen Prüfung 59
6. Programmauszug 6 Checks-Ausdruck zu Abbildung 36 60
7. Programmauszug 7 Checks-Ausdruck zu Abbildung 37 61
8. Programmauszug 8 Template der Initialisierungsroutine zur Topologieänderung des WSNs 62
9. Programmauszug 9 Definition von geschützten Bereichen innerhalb des Templates 63
10. Programmauszug 10 Einsatz von geschützten Bereichen für den manuellen Code 64
11. Programmauszug 11 Definition von geschützten Bereichen innerhalb des Templates 64
12. Programmauszug 12 Ausschnitt eines um adäquate Fehlermeldungen ergänzten Protokolls 68
13. Programmauszug 13 Einige Checks-Ausdrücke in ScatterFactory: syntaktische Randbedingungen (oben) und semantische domänenspezifische Randbedingungen (unten) 77
14. Programmauszug 14 Einsatz von Filterregeln bei der Codegenerierung 80
15. Programmauszug 15 Generierung eines alternativen Firmwaremodules 81

11.7.3. Tabellenverzeichnis

1. Tabelle 1 Die kontextfreie Grammatik 110
2. Tabelle 2 Größe des generierten Codes nach dem Kompilierungsprozess mit und ohne Filterregeln 116
3. Tabelle 3 Messung des Verarbeitungsaufwands empfangener Datenpakete in ScatterUnit 118
4. Tabelle 4 Beanspruchung der Betriebsmittel eines Sensorknotens (ESB) 120
5. Tabelle 5 Tabelleninformationen nach erneuter Testdurchführung 130
6. Tabelle 6 Vergleich der WSN Entwicklungsumgebung nach der Studie von Teng141

11.8. Literaturverzeichnis

[Agan02] Agans, D.: Debugging: The 9 Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems. Mcgraw-Hill Professional, Columbus (2002)

[Akyi02] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless sensor networks: a survey. Computer Networks, 38 (4), pp. 393--422 (2002)

[Alsa07a] Al Saad, M., Hentrich, B., Schiller, J.: ScatterFactory: An Architecture Centric Framework for Wireless Sensor Networks. In: International Conference on New Technologies, Mobility and Security, pp.12--32, Springer, Netherlands (2007)

[Alsa07b] Al Saad, M., Mysliwiec, L., Schiller, J.: ScatterPlug: A Plug-in Oriented Framework for Prototyping, Programming and Teaching Wireless Sensor Networks. In: 2nd International Conference on Systems and Networks Communications, pp. 37--47, IEEE CS Press (2007)

[Alsa07c] Al Saad, M., Ding, J., Schiller, J.: ScatterEditor: An Eclipse based Tool for Programming, Testing and Managing Wireless Sensor Networks. In: International Conference on Sensor Technologies and Applications, pp. 59--69, IEEE CS Press (2007)

[Alsa08a] Al Saad, M., Kamenzky, N., Schiller J.: Visual ScatterUnit: A Visual Model-Driven Testing of Wireless Sensor Networks Applications. In: 11th International Conference on Model Driven Engineering Languages and Systems, pp. 751--765, LNCS Vol. 5301, Springer, Berlin, Heidelberg (2008)

[Alsa08b] Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: Automated Testing of WSN Applications. In: 3rd International Conference on Systems and Networks Communications, pp. 157--167, IEEE CS Press (2008)

[Alsa08c] Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterFactory2: An Architecture Centric Framework for New Generation ScatterWeb. In: 2nd International Conference on New Technologies, Mobility and Security, 6 Pages, IEEE Xplorer (2008)

[Alsa08d] Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterClipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks. In: 6th IEEE International Symposium on Parallel and Distributed Processing and Applications, pp. 871--886, IEEE CS Press (2008)

[Baar07] Baar, M., Köppe, E., Liers, A., Schiller, J.: The ScatterWeb MSB-430 Platform for Wireless Sensor Networks (Poster and Abstract). In: SICS Contiki Hands-On Workshop, Kista, Sweden (2007)
URL: http://www.sics.se/~adam/contiki-workshop-2007/baar07scatterweb.pdf

[Bald04] Baldwin, P., Kohli, S., Lee, E. A., Liu, X., Zhao, Y.: Modeling of Sensor Nets in Ptolemy II. In: 3rd International Symposium on Information Processing in Sensor Networks, pp. 359--368, IEEE Xplore (2004)

[Blum04] Blumenthal, J., Handy, M., Timmermann, D.: SeNeTs – Test and Validation Environment for Applications in Large-Scale Wireless Sensor Networks. In: 2nd IEEE International Conference on Industrial Informatics, pp. 69--73, IEEE Xplore (2004)

[Burr06] Burri, N., Schuler, R., Wattenhofer, R.: YETI: A TinyOS Plugin for Eclipse. In:

ACM Workshop on Real-World Wireless Sensor Networks, pp.1--5, ACM, New York (2006)

[Cart06] McCartney, W. P., Nigamanth, S.: TOSDev: A rapid development environment for TinyOS. In: 4th International Conference on Embedded Networked Sensor Systems, pp. 387--388, ACM, New York (2006)

[Cheo05] Cheong, E., Lee, E. A., Zhao, Y.: Viptos: A graphical development and simulation environment for TinyOS-based wireless sensor networks. In: 3rd International Conference on Embedded Networked Sensor Systems, pp. 302--302, ACM, New York (2005)

[Corm90] Cormen, T., Leiserson, C., Rivest, R.: Introduction to Algorithms. MIT Press, Massachusetts (1990)

[Coul00] Coulouris, G., Dollimore, J. Kindberg, Tim: Distributed Systems: Concepts and Design. Addison-Wesley Longman, Amsterdam (2000)

[Cunh01] Cunha, J. C., Loureno, J., Duarte, V.: Debugging of parallel and distributed programs. In: Cunha, J. C., Kacsuk, P., Winter, S. (eds.): Parallel program development for cluster computing: methodology, tools and integrated environments (Advances in Computation: Theory and Practice, Vol. 5) pp. 97--129, Nova, Hauppauge (2001)

[Czar00] Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. Addison-Wesley Longman, Amsterdam (2000)

[Daum06a] Daum, B.: Java-Entwicklung mit Eclipse 3.2. Anwendungen, Plugins und Rich Clients. Dpunkt Verlag, Heidelberg (2006[4])

[Daum06b] Daum, B.: Das Eclipse-Codebuch: 182 Tipps, Tricks und Lösungen für Eclipse-spezifische Probleme. Dpunkt Verlag, Heidelberg (2006)

[Davi01] Johnson, D.B., Maltz, D. A., Broch, J.: DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks. In: Perkins, C. E. (ed.): Ad Hoc Networking. pp. 139--172. Addison-Wesley Longman, Amsterdam (2001)

[Ding07] Ding, J: Entwicklung einer Eclipse-Konsole für ScatterWeb. Unpublizierte Diplomarbeit, Betreuer: Mohammad Al Saad, Freie Universität Berlin (2007)

[Efft06] Efftinge, S.: Checks - Validation Language (2006)
http://www.eclipse.org/gmt/oaw/doc/4.1/r30_checkReference.pdf

[Efft08] Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M.: openArchitecture User Guide (2008)
URL: http://apps.eclipse.org/gmt/oaw/doc/4.3/openArchitectureWare-4.3-Reference.pdf

[Evan03] Evans, E.: Domain-Driven Design - Tackling Complexity in the Heart of Software. Addison-Wesley Longman, Amsterdam (2003)

[Gane03] Ganeriwal, S., Kumar, R., Srivastava, M. B.: Timing-sync protocol for Sensor Networks. In: 1st International Conference on Embedded Networked Sensor Systems, pp. 138--149. ACM, New York (2003)

[Garc08] García, E. M., Serna, M. A., Bermúdez, A., Casado, R.: Simulating a WSN-Based

Wildfire Fighting Support System. In: IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 896--902. IEEE CS Press, Washington (2008)

[Gay03] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesc language: A holistic approach to networked embedded systems. In: Conference on Programming Language Design and Implementation, pp. 1--11. ACM, New York (2003)

[Gokh04] Gokhale, A., Balasubramanian, K., Balasubramanian, J., Krishna, A., Edwards, G.T., Deng, G., Turkay, E., Parsons, J., Schmidt, D.C.: Model Driven Middleware: A new paradigm for deploying and provisioning distributed real-time and embedded applications. Journal of Science of Computer Programming: Special Issue on Foundations and Applications of Model Driven Architecture (MDA), Volume 73, Issue 1, pp. 39--58, Elsevier, Amsterdam (2008)

[Broc99] Broch, J., Johnson, D. B., Maltz, D. A.: The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks. Internet-Draft (1999)
URL: http://www.cs.cmu.edu/~dmaltz/dsr.html

[Kame07] Kamenzky, N.: Realisierung einer Testinfrastruktur für drahtlose Sensornetzwerke und Konzeption einer modellgetriebenen Testfallentwicklung. Unpublizierte Studienarbeit, Betreuer: Mohammad Al Saad, Freie Universität Berlin (2007)

[Kame08] Kamenzky, N.: Entwicklung einer modellgetriebenen Testumgebung zur Unterstützung des Testprozesses von Anwendungen drahtloser Sensornetze. Unpublizierte Diplomarbeit, Betreuer: Mohammad Al Saad, Freie Universität Berlin (2008)

[Kuor05] Kuorilehto, M., Hännikäinen, M., Hämäläinen, T. D.: A Survey of Application Distribution in Wireless Sensor Networks. In: EURASIP Journal on Wireless Communications and Networking, Issue 5, 774--788 (2005)

[Lenz07] Lenz, Ch., Chimiak-Opoka, J., Breu, R.: Model-Driven Testing of SOA-based Software. In: Luebke, D. (ed.): Workshop on Software Engineering Methods for Service-Oriented Architecture, CEUR Workshop Proceedings vol. 244, pp. 99--110. Hannover (2007)

[Levi03] Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In: 1st International Conference on Embedded Networked Sensor Systems, pp. 126--137, ACM, New York (2003)

[Losi07] Losilla, F., Vicente-Chicote, C., Álvarez, B., Iborra, A., Sánchez, P.: Wireless Sensor Network Application Development: An Architecture-Centric MDE Approach. In: Oquendo, F. (ed.): ECSA 2007, LNCS Vol. 4758, pp. 179—194. Springer, Berlin/Heidelberg (2007)

[Malt00] Maltz, D.A., Broch, J., Johnson, D.B.: Quantitative lessons from a full-scale multi-hop wireless ad hoc network testbed. In: IEEE Wireless Communications and Networking Conference, vol. 3, pp. 992--997. IEEE Press, New York (2000)

[Moor04] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.: Eclipse Development using the Graphical Editing and the Eclipse Modeling. IBM Corp. (2004)
URL: http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf

[Mysl06] Mysliwiec, L.: Entwurf und Implementierung eines Monitoring Eclipse-Frameworks für ScatterWeb. Unpublizierte Diplomarbeit, Betreuer: Mohammad Al Saad, Freie Universität Berlin (2006)

[Naum07] Naumowicz, T.: Enabling Wireless Sensor Networks: Integration of WSNs into Development Environments. Poster presented at Microsoft Research Summer School, Cambridge, UK (2007)
URL: http://page.mi.fu-berlin.de/naumowic/research/naumowicz07MRSC.pdf

[Naum08] Naumowicz, T., Freeman, R., Heil, A., Calsyn, M., Hellmich, E., Braendle, A., Guilford, T., Schiller, J.: Autonomous Monitoring of Vulnerable Habitats using a Wireless Sensor Network. In: Workshop on Real-world wireless sensor networks (European Conference on Computer Systems) pp. 51--55. ACM, New York (2008)

[Perk01] Perkins, C. E.: Ad Hoc Networking, Addison-Wesley Longman, Amsterdam (2001)

[Perk94] Perkins, C., Bhagwat, P.: Highly Dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. ACM SIGCOMM Computer Communication Review 24:4, 234--244 (1994)

[Petr06] Petrasch, R., Meimberg, O.: Model-Driven Architecture: Eine praxisorientierte Einführung in die MDA. Dpunkt Verlag, Heidelberg (2006)

[Rafi03] Rafiq O., Cacciari L.: Coordination algorithm for distributed testing. The Journal of Supercomputing 24, pp. 203--211 (2003)

[Röme04] Römer, K., Mattern, F.: The Design Space of Wireless Sensor Networks. IEEE Wireless Communications 11:6, pp. 54--61 (2004)

[Rupp05] Rupp, C., Hahn, J., Queins, S., Jeckle, M., Zengler, B.: UML 2 glasklar. Praxiswissen für die UML-Modellierung und -Zertifizierung. Hanser, München (2005)

[Rupp07] Rupp, C., Queins, S., Zengler, B.: UML 2 Glasklar, Praxiswissen für die UML-Modellierung. Hanser, München (2007^3)

[Sadi07] Sadilek, D. A.: Prototyping Domain-Specific Languages for Wireless Sensor Networks. In: Favre, J.M., Gasevic, D., Lämmel, R., Winter, A. (eds.): 4th International Workshop on Software Language Engineering (Engineering of Metamodels, Schemas, Grammars, and Ontologies, Informatik-Bericht Nr. 4) pp. 76--91. Mainz (2007)

[Sadi08] Sadilek, D. A.: Domain-Specific Languages for Wireless Sensor Networks. In: Kühne, T., Reisig, W., Steimann, F. (eds.): Modellierung 2008, LNI 127, pp. 237--241, Gesellschaft für Informatik, Bonn (2008)

[Schi05] Schiller, J., Liers, A., Ritter, H., Winter, R., Voigt, T.: ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing. In: 38th Annual Hawaii International Conference on System Sciences, vol. 9, pp. 286c, IEEE CS, Washington (2005)

[Schi07] Schiller, J.: Funksensornetze – von der Forschung zum realen Einsatz. Vortrag in der Ringvorlesung Selbstorganisierende Systeme am 26.4.07 an der Humboldt Universität.
URL: http://www2.informatik.hu-berlin.de/~daeumlic/files/rvselbstorganisierendesysteme/RV-Bericht.pdf

[Schö97] U. Schöning: Theoretische Informatik – kurzgefasst. Spektrum Akademischer Verlag, Heidelberg (1997)

[Sobe05] Sobeih, A., Chen, W., Hou, J. C., Kung, L., Li, N., Lim, H., Tyan, H., Zhang, H.: J-Sim: a simulation environment for wireless sensor networks. In: 38th Annual Simulation Symposium, pp. 175--187, IEEE Washington, DC (2005)

[Stah05] Stahl, T., Völter, M.: Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. Dpunkt Verlag, Heidelberg (2005)

[Teng08] Teng, G., Zheng, K., Dong, W.: A Survey of Available Tools for Developing Wireless Sensor Networks. In: 3rd International Conference on Systems and Networks Communications, pp. 139--144. IEEE CS Press, Washington (2008)

[Titz05] Titzer, B., Lee, D., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: 4th International Conference on Information Processing Sensor Networks, pp. 477--482, IEEE CS Press (2005)

[Ulbr05a] Ulbrich, A., Weis, T., Mühl, G., Geihs, K.: Application Development for Actuator- and Sensor-Networks. In: Römer, K. (ed.): 4. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze, Technischer Bericht TR 481, pp. 38--43. ETH Zürich (2005)
URL: http://www.vs.inf.ethz.ch/events/fg2005/fgsn05.pdf

[Ulbr05b] Ulbrich, A., Weis, T., Geihs, K.: A Modeling Language for Applications in Pervasive Computing Environments. In: 2nd Workshop on Model-based Methodologies for Pervasive and Embedded Software, pp. 109--123. TUCS, Turku (2005)
URL: http://www.tucs.fi/publications/attachment.php?fname=G39.pdf

[Ulri99] Ulrich, A. W., Zimmerer, P., Chrobok-Diening, G.: Test architectures for testing distributed systems. In: 12th International Software Quality Week, pp. 24--26. San Jose, California (1999)

[Wada07] Wada, H., Boonma, P., Suzuki, J.: Modeling and Executing Adaptive Sensor Network Applications with the Matilda UML Virtual Machine. In: 11th IASTED International Conference on Software Engineering and Applications (SEA), Cambridge, MA (2007)
URL: http://www.cs.umb.edu/~shu/papers/sea07.pdf

[Web b+m] b+m Engineering, http://www.engineering.bmiag.de/

[WebActivity] Wikipedia, Art. Activity Diagramm,
http://en.wikipedia.org/wiki/Activity_diagram

[WebAlsa] Al Saad, M.: ScatterClipse,
http://page.mi.fu-berlin.de/saad/ScatterClipse/scatterclipse.htm

[WebANT] Apache Software Foundation, Ant Projekt, http://wwwant.apache.org

[WebATL] Atlas Transformation Language (ATL), http://www.eclipse.org/m2m/atl/

[WebBart] Bartlett, J.: The art of metaprogramming. IBM (2005),
http://www.ibm.com/developerworks/linux/library/l-metaprog1.html

[WebCart] McCartney, B., Sridhar, N.: TOSDev, o.J.,
http://selab.csuohio.edu/dsnrg/tosdev/ (Link seit dem 2.8.09 nicht mehr aufrufbar)

[WebCastor] Webseite des Castor-Rahmenwerks, http://www.castor.org

[WebCDT] Eclipse C/C++ Development Tooling (CDT), http://www.eclipse.org/cdt/

[WebCheck] Check Language Reference,
http://apps.eclipse.org/gmt/oaw/doc/4.1/r30_checkReference.pdf

[WebComp] Wikipedia, Art. Komponentenbasierte Entwicklung,
http://de.wikipedia.org/wiki/Komponentenbasierte_Entwicklung

[WebCSMA-CA] Computerlexikon, CSMA-CA:
http://www.computerlexikon.com/begriff-csma-ca?highlight=Netzwerk

[Webe98] Weber, M.: Verteilte Systeme. Spektrum Akademischer Verlag, Heidelberg/Berlin (1998)

[WebEcli] Eclipse SDK, http://www.eclipse.org

[WebEcore] Meta Model (Ecore) for Describing Models,
http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/package-summary.html#details

[WebEMF] Eclipse Modeling (EMF), http://www.eclipse.org/modeling/emf/

[WebFAT] Wikipedia, Art. File Allocation Table,
http://de.wikipedia.org/wiki/File_Allocation_Table

[WebGME] Graphical Modeling Environment (GME),
http://www.isis.vanderbilt.edu/projects/gme/

[WebGMF] Graphical Modeling (GMF), http://www.eclipse.org/modeling/gmf/

[WebGRATIS] GRATIS, http://w3.isis.vanderbilt.edu/projects/nest/gratis/

[WebHome] Wikipedia, Art. Home Automation,
http://en.wikipedia.org/wiki/Home_automation

[WebICSNC] 3rd International Conference on Systems and Networks Communications,
http://www.iaria.org/conferences2008/ICSNC08.html

[WebITU] International Telecommunication Union,
http://www.itu.int/en/pages/default.aspx

[WebTTCN] Testing and Test Control Notation,
http://www.ttcn-3.org/

[WebJavaCom] Sun, The Java Communications API,
http://java.sun.com/products/javacomm/

[WebJeck]: Jeckle, M.: XML Metadata Interchange (XMI), www.jeckle.de/xmi.htm

[WebJFreeChart] JFreeChart, http://www.jfree.org/jfreechart/

[WebJUnit] Wikipedia, Art. JUnit, http://en.wikipedia.org/wiki/JUnit

[WebLog4j] Log4j-Rahmenwerk, http://logging.apache.org/log4j

[WebMDA] Model driven Architecture (MDA), http://www.omg.org/mda/

[WebMDSD] Model Driven Software Development, www.mdsd.info

[WebMOF] OMG, Meta Object Facility (MOF), http://www.omg.org/mof/

[WeboAW] OpenArchitectureWare (oAW), http://www.openarchitectureware.org/

[WeboAWDoc] oAW-Dokumentation, Version 4.2:
http://www.openarchitectureware.org/pub/documentation/4.2/html/contents/index.html

[WebOCL] OMG, Object Constraint Language Specification (OCL), v2.0,
http://www.omg.org/technology/documents/formal/ocl.htm

[WebOMG] Object Management Group (OMG), http://www.omg.org/

[WebOSGI] Seite der OSGI Allianz, http://www.osgi.org/Main/HomePage

[WebPerv] Wikipedia, Art. Pervasive Computing,
http://en.wikipedia.org/wiki/Pervasive_Computing

[WebProfiles] OMG, Profiles Spezifikation,
http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF

[WebCSMA/CA] Computerlexikon, CSMA/CA,
http://www.computerlexikon.com/begriff-csma-ca?highlight=Netzwerk

[WebRMI] Sun: Remote Methode Invocation (whitepaper),
http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[WebRMIWikipedia] Wikipedia, Art. Remote Methode Invocation,
http://de.wikipedia.org/wiki/Remote_Method_Invocation

[WebSadilek] Sadilek, D. A.: Test-Driven Language Modeling, o. J.
http://www2.informatik.hu-berlin.de/~sadilek/Homepage/Thesis_Subject.html

[WebSall] Sallai, J., Balogh, G., Dora, S.: TinyDT, Institute of Software Integrated Systems,
o.J., http://tinydt.sourceforge.net/index.html

[WebTestfall] Wikipedia, Art. Testfall,
http://de.wikipedia.org/wiki/Testfallhttp://en.wikipedia.org/wiki/Test_case

[WebTestProfile] Testing Profile, OMG,
http://www.omg.org/technology/documents/formal/test_profile.htm

[WebUML] UML, OMG, http://www.uml.org/

[WebUUID] Wikipedia, Universally Unique Identifier (UUID),

http://de.wikipedia.org/wiki/Universally_Unique_Identifier

[WebViptos] Viptos, http://ptolemy.berkeley.edu/viptos/

[WebWSN-Simu] Table comparing simulation tools for Wireless Sensor Networks, http://www.comnets.uni-bremen.de/~mab/cruise/simulation-tool-comparison-matrix.html

[WebXMI] OMG, MOF 2.0/XMI Mapping Specification, v2.1.1, OMG, o.J., http://www.omg.org/technology/documents/formal/xmi.htm

[WebXMI] XMI-Spezifikation der OMG
http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm

[WebXpand] XPand Language Reference,
http://www.openarchitectureware.org/pub/documentation/4.1/r20_xPandReference.pdf

[WebXtend] Extend Language Reference,
http://www.openarchitectureware.org/pub/documentation/4.0/r25_extendReference.pdf

[Weis91] Weiser, M.: The Computer for the Twenty-First Century. Scientific American, pp. 94--100 (1991)
URL: http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html

[Witt06] Wittenburg, G., Schiller, J.: Running Real-World Software on Simulated Wireless Sensor Nodes. In: ACM Workshop on Real-World Wireless Sensor Networks, pp.7--11, ACM, New York (2006)

[Xu04] Xu, S., Rajlich, V.: Cognitive process during program debugging. In: 3rd IEEE International Conference on Cognitive Informatics, pp. 176--182. IEEE CS Press, New York (2004)

[Zand05] Zander, J., Dai, Z. R., Schieferdecker, I., Din, G.: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In: 17th IFIP International Conference on Testing Communicating Systems, pp. 289-303, LNCS Vol. 3502, Springer, Berlin, Heidelberg (2005)

11.9. Aus der Arbeit entstandene Publikationen

Journals/Book Chapters:

Al Saad, M., Schiller, J., Fehr, E.:
Automated Testing and Development of WSN Applications. Book chapter for "Wireless Sensor Network", ISBN 978-3-902613-49-3, to be published by IN-TECH, end of 2009

Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.:
Model-Driven Visual Testing and Debugging of WSN Applications
In: Journal of Networks (JNW), VOL. 4, NO. 7, Academy Publisher. (2009)

Proceedings - Conferences/Workshops[21]:

Al Saad, M., Hentrich, B., Schiller, J.: ScatterFactory: An Architecture Centric for Wireless Sensor Networks. In: International Conference on New Technologies, Mobility and Security, pp.12--32, Springer, Netherlands (2007)

Al Saad, M., Mysliwiec, L., Schiller, J.: ScatterPlug: A Plug-in Oriented for Prototyping, Programming and Teaching Wireless Sensor Networks. In: 2nd International Conference on Systems and Networks Communications, pp.37--47, IEEE CS Press (2007)

Al Saad, M., Ding, J., Schiller, J.: ScatterEditor: An Eclipse based Tool for Programming, Testing and Managing Wireless Sensor Networks. In: International Conference on Sensor Technologies and Applications, pp.59--69, IEEE CS Press (2007) ***(Best Paper Award)***

Al Saad, M., Kamenzky, N., Schiller J.: Visual ScatterUnit: A Visual Model-Driven Testing of Wireless Sensor Networks Applications. In: 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 751--765, LNCS Vol. 5301, Springer, Berlin, Heidelberg (2008)

Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: Automated Testing of WSN Applications. In: 3nd International Conference on Systems and Networks Communications, pp.157--167, IEEE CS Press (2008)

Al Saad, M., Fehr, E., Kamenzky, N., Schiller, J.: ScatterClipse: A Model-Driven Tool-Chain for Developing, Testing, and Prototyping Wireless Sensor Networks.
In: 6th IEEE International Symposium on Parallel and Distributed Processing and Applications, pp. 871--886, IEEE CS Press (2008)

[21] Jede der aufgelisteten Veröffentlichungen wurde von mir in der jeweiligen Konferenz präsentiert.